JavaScript and JSON Essentials,Second Edition

# JavaScript与JSON从入门到精通（第2版）

[美] 布鲁诺·约瑟夫·德梅洛 等 著　　刘晓雪 译



清华大学出版社

# JavaScript 与 JSON 从入门到精通

（第 2 版）

[美] 布鲁诺·约瑟夫·德梅洛　等著
刘晓雪　译

清华大学出版社
北　京

## 内 容 简 介

本书详细阐述了与 JSON 相关的基本解决方案，主要包括 JSON 简介、JSON 结构、基于 JSON 的 AJAX 请求、跨域异步请求、JSON 调试、构建 Carousel 应用程序、JSON 的替代方案、hapi.js 简介、在 MongoDB 中存储 JSON 文档、利用 JSON 配置任务管理器、实时系统和分布式系统中的 JSON、JSON 用例等内容。此外，本书还提供了相应的示例、代码，以帮助读者进一步理解相关方案的实现过程。

本书既可作为高等院校计算机及相关专业的教材和教学参考书，也可作为相关开发人员的自学教材和参考手册。



北京市版权局著作权合同登记号　图字：01-2019-1270



**图书在版编目（CIP）数据**

JavaScript 与 JSON 从入门到精通：第 2 版/（美）布鲁诺 • 约瑟夫 • 德梅洛等著；刘晓雪译. —北京：清华大学出版社，2019

书名原文：JavaScript and JSON Essentials-Second Edition
ISBN 978-7-302-53242-2

I. ①J… II. ①布… ②刘… III. ①JAVA 语言-程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字（2019）第 131108 号

**责任编辑：** 贾小红
**封面设计：** 刘　超
**版式设计：** 文森时代
**责任校对：** 马军令
**责任印制：** 刘海龙

**出版发行：** 清华大学出版社
网　址：http://www.tup.com.cn，http://www.wqbook.com
地　址：北京清华大学学研大厦 A 座　　邮　编：100084
社 总 机：010-62770175　　邮　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
**印 装 者：** 三河市国英印务有限公司
**经　销：** 全国新华书店
**开　本：** 185mm×230mm　　**印　张：** 10.75　　**字　数：** 218 千字
**版　次：** 2019 年 7 月第 2 版　　**印　次：** 2019 年 7 月第 1 次印刷
**定　价：** 89.00 元

---

产品编号：082572-01

# 译　者　序

JSON 和 XML 是较为流行的数据交换格式，在 Web API、NoSQL 数据库、服务端编程语言和客户端框架中都可以看到 JSON 的身影。在不同平台间的传递数据方面，JSON 已成为 XML 强有力的替代者。与 XML 相比，JSON 更加简洁且易于阅读，同时方便检查排错。另外，JSON 更加轻量级，不管是编写、传输，还是解析都更加高效。JSON 在传输过程中采用了压缩技术，因而更加节省带宽。最后，JSON 还得到了众多语言的支持，如 JavaScript、Python、C、C++等主流语言。

本书通过示例展示了 JSON 在 Web 开发中扮演的不同角色。在了解了 JSON 的基础知识后，将在 Angular 5、Node.js、模板嵌入机制的基础上实现 JSON 应用程序。对于服务器端的脚本编程，本书还将尝试实现 Hapi.js（因其可配置的 JSON 架构著称）。此外，本书还将学习如何使用 Kafka 为实时应用程序实现 JSON，以及如何为任务运行器和 MongoDB BSON 存储实现 JSON。本书通过 JSON 格式的案例进行讲解，帮助读者构建快速、可伸缩和高效的 Web 应用程序。

在本书的翻译过程中，除刘晓雪外，王辉、刘璋、张博、刘祎、张华臻等人也参与了部分翻译工作，在此一并表示感谢。

由于译者水平有限，难免有疏漏和不妥之处，恳请广大读者批评指正。

译　者

# 前　　言

JSON 是数据交换的一种标准格式，本书将通过各种示例讨论 JSON 在 Web 开发中饰演的不同角色。在阅读完本书后，读者将会以全新的角度理解应用程序的解决方案和复杂问题的处理方式。

## 适用读者

如果读者是一名对 JavaScript 或 PHP 开发有着基本了解的 Web 人员，并且希望编写 JSON 数据进而将其与 RESTful API 集成，以创建快速、可伸缩的应用程序，那么，本书将十分适合于您。

## 本书内容

**第 1 章：JSON 简介**。将讨论 JSON 的历史及其工作方式和内存中的存储方式。另外，本章还将介绍一些支持 JSON 的、较为流行的编程语言。在本章结束时，还将利用不同的 JSON 数据类型编写一个较为基础的应用程序。

**第 2 章：JSON 结构**。将利用多种数据类型、多个对象和多维数据进一步丰富 JSON 实现。

**第 3 章：基于 JSON 的 AJAX 请求**。将探讨基于 JSON 数据的 AJAX 请求，并通过 HTTP 请求传递 JSON 数据，以及处理此类问题的异步技术。

**第 4 章：跨域异步请求**。介绍跨域的异步调用这一概念。由于数据将在域间进行传输，因而用户有必要了解基于填充（padding）的 JSON 设疑概念，即 JSONP。

**第 5 章：JSON 调试**。将讨论可用于调试、验证和格式化 JSON 的强大工具。

**第 6 章：构建 Carousel 应用程序**。实现了 Carousel 应用程序的编程思想，以及应用程序所需的设置项和依赖项，如 jQuery 库和 jQuery Cycle 插件，并使用 Bootstrap 来维护应用程序的基本设计。

**第 7 章：JSON 的替代方案**。讨论了 JSON 的非 Web 开发实现，如依赖项管理器、元数据存储和配置存储。

**第 8 章：hapi.js 简介**。介绍在 Hapi 服务器中实现基于 JSON 的配置，并借助于 Hapi

创建 RESTful API。

**第 9 章：在 MongoDB 中存储 JSON 文档**。讨论 MongoDB，以及 JSON 在 MongoDB 中的使用方式。随后，本章还将介绍如何在 MongoDB 文档上执行不同的操作。

**第 10 章：利用 JSON 配置任务管理器**。将简要描述 gulp.js 库。Gulp 是一个功能强大的库，主要用于构建任务的管理并提供相关工具。

**第 11 章：实时系统和分布系统中的 JSON**。通过实现 socket.io 服务器，使读者熟悉 JSON 数据在实时 Web 应用程序中的应用，以及 Apache Kafka。

**第 12 章：JSON 中的用例**。将讨论一个用例，并考查 JSON 针对不同领域的增强方案，以及移植后 JSON 所提供的各种优点。

## 阅读方式

如果读者是一名 Web 开发的初学者，可从第 1 章开始阅读，并了解 JSON 中的基础知识。另外，前 5 章简单易懂且便于操作。在后续学习过程中，读者可尝试实现每章所提供的代码片段。

随着时间的推移，读者还可在 StackOverflow 或 GitHub 等论坛上进行讨论，以确保书中的所有问题均已被解决。

## 软件环境和资源下载

读者可访问 http://www.packtpub.com 并通过个人账户下载示例代码文件。另外，在 http://www.packtpub.com/support 中注册成功后，我们将以电子邮件的方式将相关文件发与读者。

读者可根据下列步骤下载代码文件。

- ❑ 利用电子邮件和密码登录或注册我们的网站 www.packtpub.com。
- ❑ 单击 SUPPORT 选项卡。
- ❑ 单击 Code Downloads & Errata。
- ❑ 在 Serach 文本框中输入书名。

当文件下载完毕后，确保使用下列最新版本软件解压文件夹。

- ❑ Windows 系统下的 WinRAR/7-Zip。
- ❑ Mac 系统下的 Zipeg/iZip/UnRarX。
- ❑ Linux 系统下的 7-Zip/PeaZip。

另外，读者还可访问 GitHub 获取本书的代码包，对应网址为 https://github.com/

PacktPublishing/JavaScript-and-JSON-Essentials-Second-Edition。

此外，读者还可访问 https://github.com/PacktPublishing/以了解丰富的代码和视频资源。

最后，读者还可访问 https://www.packtpub.com/sites/default/files/downloads/JavaScriptandJSONEssentialsSecondEdition_ColorImages.pdf 以下载并查看书中的图片。

## 本书约定

本书通过不同的文本风格区分相应的信息类型。下面通过一些示例对此类风格以及具体含义的解释予以展示。

代码块如下所示。

```
for(let j=0;j<designationCount;j++){
  designations+= `, ${data json[i].designation.title[j]}`;
}
```

当某个代码块希望引起读者的足够重视时，一般会采用黑体表示，如下所示。

```
const http = require('http');
const port = 3300;
http.createServer((req, res) => {
  res.writeHead(200, {
    "Content-Type": "application/json"
  });
  res.write(JSON.stringify({
    greet : "Hello Readers!"
  }));
  res.end();
}).listen(port);
console.log(`Node Server is running on port : ${port}`)
```

命令行输入或输出则采用下列方式表达。

```
$ mkdir test-node-app
$ cd test-node-app
$ npm init
```

图标则表示较为重要的说明事项。

图标则表示提示信息和操作技巧。

## 读者反馈和客户支持

欢迎读者对本书的建议或意见予以反馈。

对此，读者可向 feedback@packtpub.com 发送邮件，并以书名作为邮件标题。若读者对本书有任何疑问，均可发送邮件至 questions@packtpub.com，我们将竭诚为您服务。

若读者针对某项技术具有专家级的见解，抑或计划撰写书籍或完善某部著作的出版工作，则可访问 www.packtpub.com/authors。

## 勘误表

尽管我们在最大程度上做到尽善尽美，但错误依然在所难免。如果读者发现谬误之处，无论是文字错误抑或是代码错误，还望不吝赐教。对此，读者可访问 http://www.packtpub.com/submit-errata，选取对应书籍，单击 ErrataSubmissionForm 超链接，并输入相关问题的详细内容。

## 版权须知

一直以来，互联网上的版权问题从未间断，Packt 出版社对此类问题异常重视。若读者在互联网上发现本书任意形式的副本，请告知网络地址或网站名称，我们将对此予以处理。关于盗版问题，读者可发送邮件至 copyright@packtpub.com。

## 问题解答

若读者对本书有任何疑问，均可发送邮件至 questions@packtpub.com，我们将竭诚为您服务。

# 目　　录

# 第 1 章　JSON 简介

JSON（JavaScript 对象表示法）是一种非常流行的数据交换格式，最先由 Douglas Crockford 爵士发布。根据 Douglas Crockford 所言，JSON 通常以对象表示法而存在。另外，Douglas Crockford 并非发明了 JSON，但他首次制定了 JSON 规范并设计了 JSON，以便将其用作标准格式。

本章主要涉及以下主题。

- 什么是 JSON?
- 利用 JSON 实现简单的 Hello World 程序。
- JSON 中的数据类型。
- JSON 所支持的各种语言。
- PHP 和 Python 简介。

对于第一次听说 JSON 这一术语的所有初学者来说，下面的各个小节将帮助您了解 JSON。

## 1.1　数据交换格式 JSON

对于客户端和服务器间的数据交换，当定义 JSON 时，我们可以说这是一种基于文本的、轻量级的、具有人类可读的数据格式。JSON 源于 JavaScript，与 JavaScript 对象非常相似，但独立于 JavaScript。另外，JSON 独立于任何语言，但所有流行的编程语言均对 JSON 数据格式提供了支持，其中包括 C#、PHP、Java、C++、Python 和 Ruby。

JSON 可用于 Web 应用程序的数据交换行为。考查如图 1.1 所示的简单的客户端-服务器架构。假设客户端为浏览器，并向服务器发送 HTTP 请求；随后，服务器按预期处理请求并提供响应结果。

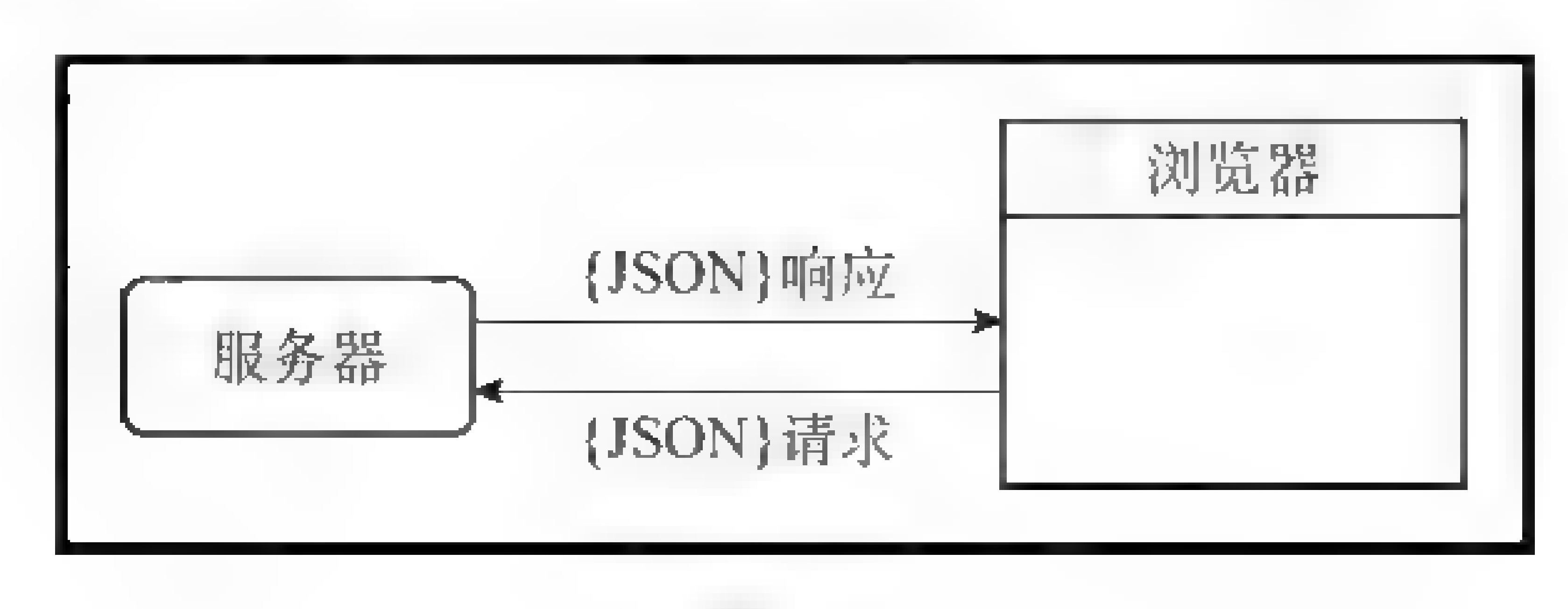



图　1.1

在上述双向通信中，所用的数据格式表示为一个序列化的字符串，且括号中包含了键-值对组合——这便是 JSON。

在 JSON 之前，XML 曾被视作所选的数据交换格式。XML 解析需要客户端的 XML DOM 实现来接收 XML 响应，然后使用 XPath 查询响应以访问和检索数据。这一过程较为烦琐，数据查询须在两级上执行：首先是服务器端，其间，数据通过数据库被查询；其次是在客户端使用 XPath。相比之下，JSON 则不需要特定的实现，浏览器中的 JavaScript 引擎则负责处理 JSON 的解析工作。

在通过网络连接发送数据时，XML 消息通常比较繁重和冗长，并占用大量带宽。一旦检索到 XML 消息，该消息需要加载至内存中以被处理。下面分别考查 XML 格式和 JSON 格式的 students 数据传输。

下列代码展示了 XML 格式示例。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- This is an example of student feed in XML -->
<students>
  <student>
    <studentid>101</studentid>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
    <classes>
      <class>Business Research</class>
      <class>Economics</class>
      <class>Finance</class>
    </classes>
  </student>
  <student>
    <studentid>102</studentid>
    <firstname>Jane</firstname>
    <lastname>Dane</lastname>
    <classes>
      <class>Marketing</class>
      <class>Economics</class>
      <class>Finance</class>
    </classes>
  </student>
</students>
```

下列代码展示了 JSON 格式示例。

```
/* This is an example of student feed in JSON */
{
```

```
"students": {
  "0": {
    "studentid": 101,
    "firstname": "John",
    "lastname": "Doe",
    "classes": [
      "Business Research",
      "Economics",
      "Finance"
    ]
  },
  "1":{
    "studentid": 102,
    "firstname": "Jane",
    "lastname": "Dane",
    "classes": [
      "Marketing",
      "Economics",
      "Finance"
    ]
  }
 }
}
```

需要注意的一点是，当与 JSON 相比时，XML 消息尺寸较大，而此处仅展示了两个记录。实时传入数据至少包含数千个记录。需要注意的另一点是，由服务器生成并通过互联网传输的数据量已经很大，而 XML 的冗长使得数据量变得更大。在移动设备时代，智能手机和平板电脑日益流行，在速度较慢的网络上传输大量数据将会导致页面加载缓慢、死机、较差的用户体验和访问量的降低。目前，JSON 已经成为首选的互联网数据交换格式，以避免前面提到的各类问题。由于 JSON 用于在互联网上传输序列化的数据，因此我们需要了解其 MIME 类型。

图 1.2 显示了请求数据如何发送至之前提到的客户端-服务器架构中。

多用途互联网邮件扩展（Multipurpose Internet Mail Extensions，MIME）类型是一种互联网媒体类型，这是由两部分构成的内容标识符，进而在互联网上传输。MIME 类型通过 HTTP 请求和 HTTP 响应的 HTTP 头文件传递。MIME 类型可视为服务器和浏览器间的内容类型通信。总体来说，MIME 类型将包含两个或多个部分，并提供与数据类型（在 HTTP 请求或 HTTP 响应中发送）相关的浏览器信息。JSON 数据的 MIME 类型表示为应用程序/json。如果 MIME 类型头文件没有通过浏览器发送，它会将传入的 JSON 视为纯文本。

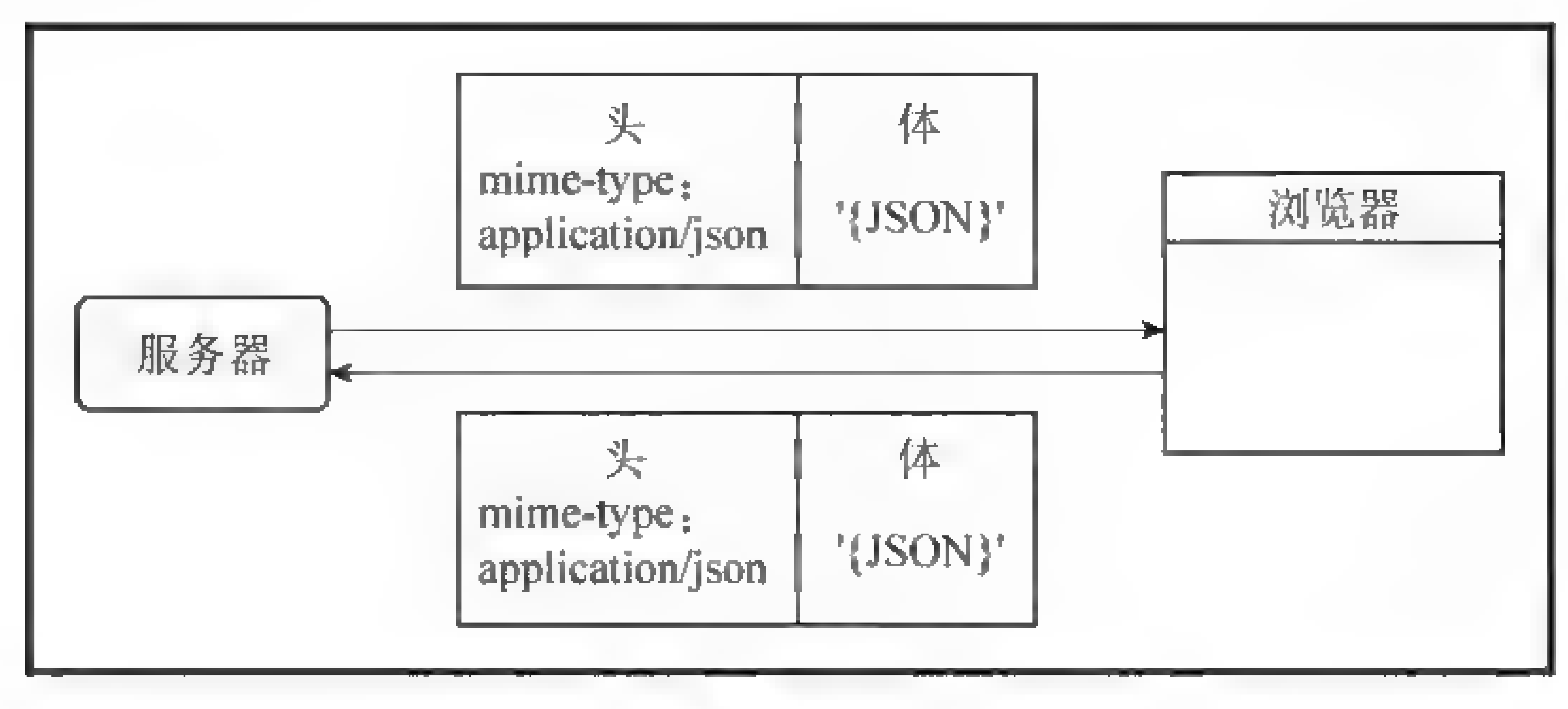



图 1.2

现在，content-type 键（派生自 mime-type 自身）可用于数据头中

## 1.2　基于 JSON 的 Hello World 程序

前述内容介绍了 JSON 的基本知识，接下来像往常一样编写一个 Hello World 应用程序，对应的代码片段如下所示。

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Javascript</title>
    <script type="text/javascript">
      let hello world = {"Hello":"World"};
      alert(hello world.Hello);
    </script>
  </head>
  <body>
    <h2>JSON Hello World</h2>
    <p>This is a test program to alert Hello world!</p>
  </body>
</html>
```

当在浏览器中被调用时，上述程序将在屏幕上显示 Hello World。

此处使用了新的 ECMAScript 标识符 let，其作用域与一般的变量声明标识符 var 有所不同。前者的作用域是最近的函数块，而后者的作用域则是最近的封闭块。对此，读者可访问 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let 以了解更多信息。

此处应特别关注<script>标签之间的脚本，如下所示。

```
let hello_world = {"Hello":"World"};
alert(hello_world.Hello);
```

此处首先生成一个 JavaScript 变量，并利用 JavaScript 对象初始化该变量。类似于从 JavaScript 对象中检索数据，我们也可采用键-值对检索数值。简单地讲，JSON 表示为一个键-值对集合。其中，每个键表示为数值在计算机上的存储位置的引用。下面让我们思考一下，如果全部工作是分配 JavaScript 对象，为何我们需要使用 JSON。答案在于，JSON 是一种完全不同的格式，而不像 JavaScript 那样是一种语言。

ⓘ JSON 键和值必须用双引号括起来。如果任何一个都包含在单引号中，则将会得到一条错误消息。

接下来快速浏览一下 JSON 和 JavaScript 对象之间的相似性和差异。如果打算创建一个与之前 hello_world JSON 示例类似的 JavaScript 对象，对应代码如下所示。

```
let hello_world = {"Hello":"World"};
```

此处的差别在于，键并未包含于双引号中。由于 JSON 键定义为一个字符串，因而可对其使用任意有效的字符串。例如，可在键中使用空格、特殊字符和连字符，则这在常规的 JavaScript 对象中均是无效的，如下所示。

```
let hello_world = {"test-hello":"World"};
```

当在键中使用特殊字符、连字符或空格时，我们在对其进行访问时须格外小心，如下所示。

```
alert(hello_world.test-hello); //doesn't work
```

上述 JavaScript 语句无法正常工作的原因在于，JavaScript 不接受包含特殊字符、连字符或字符串的键。因此，需要通过某种方法检索数据，在这个方法中，我们将 JSON 对象作为一个包含字符串键的关联数组来处理，如下所示。

```
alert(hello_world["test-hello"]); //Hurray! It work
```

二者间的另一个差别在于，JavaScript 对象可在内部携带函数，而 JSON 对象则无法携带任何函数。下列示例设置了一个属性 getFullName，其中包含了一个函数并在被调用时显示名称 John Doe。

```
{
  "studentid": 101,
  "firstname": "John",
```

```
  "lastname": "Doe",
  "isStudent": true,
  "classes": [
    "Business Research",
    "Economics",
    "Finance"
  ],
  "getFullName": function(){
   alert(`${this.firstname} ${this.lastname}`);
  }
}
```

注意，字符串插值特性是一个新的 ES 特性，可以在表达式“`${}`”中编写变量和函数时使用，该表达式只适用于波浪引号（即在键盘波浪线“～”键下方的引号“`”），而不适用于其他类型的引号

最后，二者间最大的差别在于，JavaScript 对象并不是一种数据交换格式，而 JSON 的唯一的目的即用作一种数据交换格式。

接下来将讨论 JSON 的内存分配。

## 1.3　如何在内存中存储 JSON

本节主要在 JSON 和 JavaScript 对象之间进行比较。如前所述，JSON 是对象的字符串化表达，为了从概念上理解 JSON 的存储过程，考查下列示例代码。

```
let completeJSON = {
  "hello": "World is a great place",
  "num": 5
}
```

下面将在完整的 JSON 上执行相关操作，进而对这一概念进行划分。这里，完整的 JSON 是指一类整体结构，而不是键-值对的子集。因此，第一项操作是序列化。序列化是删除空格以及转义内部引号（如果存在）的过程，以使将整个结构转换为单个字符串，如下所示。

```
let stringifiedJSON = JSON.stringify(completeJSON);
```

如果在控制台输出变量 stringifiedJSON，对应结果如下所示。

```
"{"hello":"World is a great place","num":5}"
```

在当前示例中，JSON 全部存储为常规的字符串，如图 1.3 所示。




图　1.3

在上述概念图 1.3 中，内存位置的 slot 1 通过包含单一输入字符串值的 stringifiedJSON 表示。下一个内存位置的存储类型可能是解析的 JSON。通过前述代码片段，如果对 stringifiedJSON 进行解析，如下所示。

```
let parsedJSON = JSON.parse(stringifiedJSON);
```

对应的输出结果如下所示。

```
{
  "hello": "World is a great place",
  "num": 5
}
```

在上述 JSON 表达结果中，可以清楚地看到，所得到的值是一个对象，而不是一个字符串。进一步讲，这是一种 JavaScript 对象表示法。当前，在内存中存储 JSON 这一概念与 JavaScript 对象相同。

假设我们使用 JavaScript 引擎解释代码。

在当前示例中，对应场景则完全不同，如图 1.4 所示。

在考查了内存表达方式之后，可以得到以下结论以供参考。

- 对象存储不是顺序的；内存插槽是随机选择的。在当前示例中是 slot 4 和 slot 7。
- 存储在每个插槽中的数据是对不同内存位置的引用，并可将其称作地址。

在当前示例中，我们可得到包含地址 object.hello 的第 4 个内存插槽。目前，该地址指向不同的内存位置。假设对应位置为第 3 个内存插槽，并被 JavaScript 执行上下文所处理。因此，parsedJSON.hello 值被第 3 个内存插槽所持有。

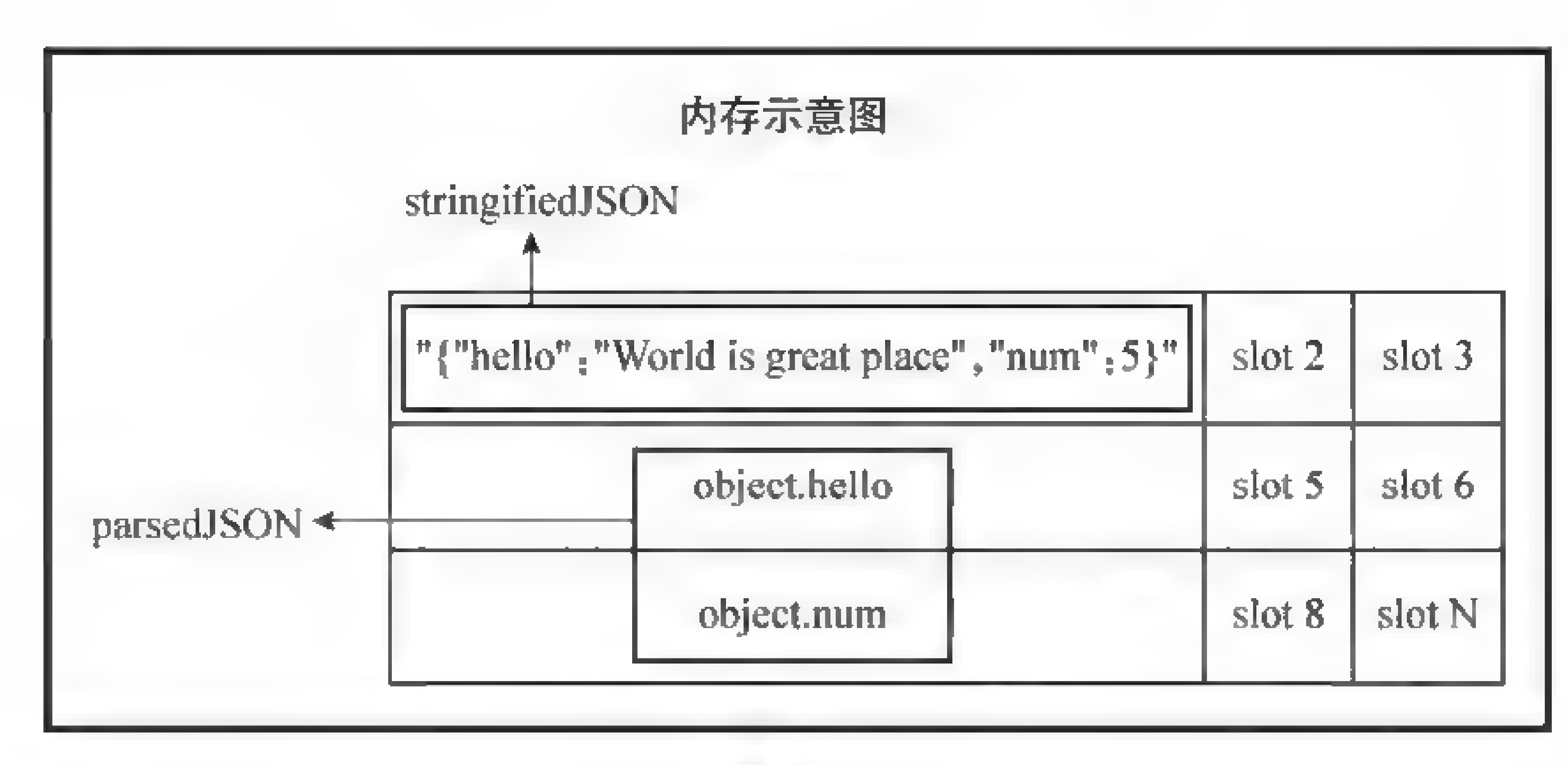



图　1.4

## 1.4　JSON 的数据类型

本节将讨论更加复杂的 JSON 示例，并介绍 JSON 所支持的全部数据类型。具体来说，JSON 支持 6 种数据类型，即字符串、数字、布尔值、数组、对象和 null，如下所示。

```
{
  "studentid": 101,
  "firstname": "John",
  "lastname": "Doe",
  "isStudent": true,
  "scores": [40, 50],
  "courses": {
    "major": "Finance",
    "minor": "Marketing"
  }
}
```

上述示例中包含了不同数据类型的键-值对，下面对此逐一加以讨论。

- "studentid"引用的值的数据类型表示为一个数字，如下所示。

```
"studentid": 101,
```

- "firstname"引用的值的数据类型是一个字符串，如下所示。

```
"firstname": "John",
```

- 在随后的代码片段中，"isStudent"引用的值的数据类型为一个布尔值，如下所示。

```
"isStudent": true,
```

- "scores"引用的值的数据类型为一个数组，如下所示。

```
"scores": [40, 50]
```

- "courses"引用的值的数据类型为一个对象，如下所示。

```
"courses": {
  "major": "Finance",
  "minor": "Marketing"
}
```

如前所述，JSON 支持 6 种数据类型，分别为字符串、数字、布尔值、数组、对象和 null。需要注意的是，JSON 支持 null 数据，且实时业务实现需要准确的信息。某些时候，可能会出现空字符串替换 null 的情况，但这是不准确的，接下来考查一个示例，如下所示。

```
let nullVal = "";
alert(typeof nullVal); //prints string
nullVal = null;
alert(typeof nullVal); //prints object
```

数组和 null 值均为 JavaScript 中的对象

在上述示例中，我们执行了一些简单的操作，例如确定空字符串的类型。此处采用了 typeof 运算符，它接收一个操作数并返回该操作数的数据类型；而下一行则确定 null 值的类型。

下列代码实现了 JSON 对象并对值进行检索。

```
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript">
    let complexJson = {
      "studentid": 101,
      "firstname": "John",
      "lastname": "Doe",
      "isStudent": true,
      "scores": [40, 50],
      "courses": {
        "major": "Finance",
        "minor": "Marketing"
      }
    };
  </script>
</head>
```

```
<body>
  <h2>Complex Data in JSON</h2>
  <p>This is a test program to load complex json data into a variable</p>
</body>
</html>
```

当从变量 complexJson 中检索 id 时，需要执行下列代码。

```
alert(complexJson.studentid); //101
```

当从 complexJson 变量中检索 name 时，需要执行下列代码。

```
alert(complexJson.firstname); //John
```

下列代码则从变量 complexJson 中检索 isStudent。

```
alert(complexJson.isStudent); //true
```

相比较而言，从数组和对象中获取数据则稍显复杂——必须遍历数组或对象。下列代码展示了如何从数组中检索值。

```
alert(complexJson.scores[1]); //50
```

上述代码从 scores 数组中检索第 2 个元素（数组始于 0，因而其索引为 1）。虽然 scores 在 complexJson 中表示为一个数组，但仍视为一个常规的键-值对。当访问键时，须采用不同方式对此进行处理。在访问键时，解释器首先需要评估如何获取其值的数据类型。如果检索到的值为字符串、数字、布尔值或 null，则在该值上无须执行进一步的操作；而对于数组或对象，还须进一步考查该值的依赖关系。

当从 JSON 对象内部的对象中检索某个元素时，需要访问该值所引用的键，如下所示。

```
alert(complexJson.courses.major); //Finance
```

由于对象不包含数值索引，JavaScript 将重新排列对象中数据项的顺序。读者可能已经注意到，JSON 对象初始化阶段的键-值对顺序与访问时的顺序有所不同，但我们不必对此过虑。在这期间并不存在数据丢失，JavaScript 仅对对象重新排序而已。

## 1.5　支持 JSON 的编程语言

前述内容讲述了 JavaScript 中的解释器如何支持 JSON。除此之外，还存在其他一些编程语言对 JSON 实现提供了支持。例如 PHP、Python、C#、C++、Java 均对 JSON 数据交换格式提供了较好的支持。所有支持面向服务架构的流行编程语言都理解 JSON 及其数据传输实现的重要性，因此为 JSON 提供了强大的支持。

接下来我们来看一下 JSON 在其他语言（例如 PHP 和 Python）中是如何实现的。

### 1.5.1　PHP 中的 JSON 实现

就构建 Web 应用程序来说，PHP 也是一种较为流行的语言，该语言是一种服务器端脚本语言，以使开发人员能够构建应用程序，并在服务器上执行相关操作、连接数据库以执行 CRUD（创建、读取、更新和删除）操作，同时对实时应用程序提供安全的环境。自 PHP 5.2.0 起，JSON 即构建于 PHP 内核中，这也使得用户不必执行复杂的安装和配置操作。考虑到 JSON 仅是一种数据交换格式，PHP 中包含了两个功能项，分别成立通过请求传入的 JSON，或生成通过响应发送的 JSON。PHP 是一种弱类型语言，因而我们将使用存储于 PHP 数组中的数据，并将该数据转换为 JSON 字符串，进而用于数据输入。下面再次考查之前讨论的 student 示例，并在 PHP 中构建，随后将其转换为 JSON。

注意，当前示例仅用于展示利用 PHP 生成 JSON

```
<?php
 $student = array(
   "id"=>101,
   "name"=>"John Doe",
   "isStudent"=>true,
   "scores"=>array(40, 50);
   "courses"=>array(
     "major"=>"Finance",
     "minor"=>"Marketing"
   );
 );

 //Echo is used to print the data
 echo json_encode($student); //encoding the array into JSON string

?>
```

当运行 PHP 脚本时，需要安装 PHP；当通过浏览器运行 PHP 脚本时，则需要使用到一个 Web 服务器，例如 Apache 或 IIS。在第 3 章中与 AJAX 协同工作时，将详细讨论安装过程。

对应的脚本在启动过程中将初始化变量，并分配包含学生信息的关联数组。随后，变量$students 将被传递至 json_encode()函数中，该函数将该变量转换为 JSON 字符串。当运行脚本时，将生成一个有效的响应结果，并将其公开为 JSON 数据输入，以供其他应用程序使用。

对应的输出结果如下所示。

```
{
  "id": 101,
  "name": "John Doe",
  "isStudent": true,
  "scores": [40, 50],
  "courses":
  {
    "major": "Finance",
    "minor": "Marketing"
  }
}
```

至此，我们通过简单的 PHP 脚本生成了第一个 JSON。其中，对应方法解析通过 HTTP 请求传入的 JSON，对于发送 HTTP 请求并以 JSON 格式发送数据的应用程序来说，这种情况十分常见。

**该示例仅展示如何将 JSON 导入 PHP 中**

```
$student = '{"id":101,"name":"John
  Doe","isStudent":true,"scores":[40,50], "courses":{"major": "Finance",
  "minor":"Marketing"}}';
//Decoding JSON string into php array
print_r(json_decode($student));
```

对应的输出结果如下所示。

```
Object(
  [id] => 101
  [name] => John Doe
  [isStudent] => 1
  [scores] => Array([0] => 40[1] => 50)
  [courses] =>stdClass
  Object([major] => Finance[minor] => Marketing)
)
```

### 1.5.2　Python 中的 JSON 实现

Python 是一种非常流行的脚本语言，被广泛地应用于执行字符串操作和构建控制台应用程序中。Python 可从 JSON API 中获取数据，一旦检索到 JSON 数据，对应结果将被视为 JSON 字符串。当在 JSON 字符串上执行任意操作时，Python 提供了相应的 JSON 模块。JSON 模块是许多功能强大的函数的组合，可用于解析 JSON 字符串。

当前示例仅用于展示如何利用 Python 生成 JSON。

```
import json

student = [{
  "studentid": 101,
  "firstname": "John",
  "lastname": "Doe",
  ## make sure we have first letter capitalize in case of boolean
  "isStudent": True,
  "scores": [40, 50],
  "courses": {
    "major": "Finance",
    "minor": "Marketing"
  }
}]

print json.dumps(student)
```

该示例中使用了较为复杂的数据类型，例如元组和字典，分别用于存储分数值和课程。此处并不打算对 Python 的数据类型予以深入讨论。

当运行上述脚本时，需要安装 Python 2。在任何*nix 操作系统上，均预安装了 Python 2。目前，读者可访问 https://www.jdoodle.com/pythonprogramming-online，并利用在线执行器运行当前代码

对应的输出结果如下所示。

```
[{"studentid": 101, "firstname": "John", "lastname": "Doe", "isStudent":
true, "courses": {"major": "Finance","minor": "Marketing"}, "scores":
[40,50]}]
```

其中，键可能会根据数据类型重新排序。另外，也可使用 sort_keys 检索原始顺序。下面简要介绍一下 JSON 在 Python 中的解码方式。

该示例仅用于展示如何将 JSON 导入 Python 中。

```
student_json = '''[{"studentid": 101, "firstname": "John", "lastname":
"Doe", "isStudent": true, "courses": {"major": "Finance", "minor":
"Marketing"}, "scores": [40, 50]}]'''

print json.loads(student_json)
```

其中，JSON 字符串存储至 student_json 中，同时使用了 Python 中 JSON 模块提供的

json.loads()方法。

对应的输出结果如下所示。

```
[
{
 u 'studentid': 101,
 u 'firstname': u 'John',
 u 'lastname': u 'Doe',
 u 'isStudent': True,
 u 'courses':
 {
   u 'major': u 'Finance',
   u 'minor': u 'Marketing'
 },
 u 'scores': [40, 50]
}]
```

## 1.6　本 章 小 结

本章介绍了 JSON 方面的基础知识、JSON 的历史以及 JSON 的优点（相比于 XML）。其间，我们还创建了第一个 JSON 对象，并对其进行了成功的解析。另外，本章还讨论了 JSON 所支持的所有数据类型。最后，我们通过一些示例展示了 JSON 在其他语言中的实现方式。本章内容为后续章节中将要学习的复杂概念（例如访问多级 JSON，并在其上执行数据存储操作）奠定坚实的基础。

# 第 2 章　JSON 结构

第 1 章讨论了 JSON 的基础知识、JSON 如何嵌入 HTML 文件中，以及如何在简单的 JSON 对象上执行基本操作，例如键的访问。本章主要涉及以下主题。

- 插入外部 JavaScript。
- 访问多级 JSON 对象。
- 在 JSON 数据中执行修改操作。

现在让我们向前迈进一步，并使用更大、更复杂、更接近于现实世界中使用的 JSON 对象。

## 2.1　插入外部 JavaScript

在现实的应用程序中，JSON 可作为异步请求的响应结果，或者从 JSON 提要（feed）中被检索。网站一般会通过 HTML、CSS 和 JavaScript 提供精美的视觉化用户界面。但有些情况下，数据供应商只专注于获取数据，而数据提要则用于这一目的。整体来讲，提要是提供数据的一种粗糙方式，其他人可对其加以复用，以在网站上显示数据；或者获取数据并在其上执行相关算法。这一类数据提要一般较大，且无法直接嵌入脚本标签中。接下来讨论如何在 HTML 文件中包含外部 JavaScript 文件。

以下内容显示了 external-js.html 文件中的代码。

```
<!DOCTYPE html>
<html>
<head>
  <title>Include external javascript</title>
  <script type="text/javascript" src="example.js"></script>
  <script type="text/javascript">
    alert(x);
  </script>
</head>
<body>
  <h2>Include external javascript</h2>
  <p>This is a test program to learn how external javascript files can
    be included</p>
</body>
</html>
```

当前示例中包含了 example.js 这一外部 JavaScript 文件，如下所示。

```
const x = "This is value of x and is being retrieved from external js
file";
```

注意，const 关键字用作变量标识符以表示不可变数据　一旦声明后，常量 x 便无法被重新赋值　对此，读者可访问 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const 以了解更多内容。

当从 external-js.html 文件中访问位于 example.js 文件中的常量 x 时，需要在 HTML 文件的脚本标签中编写程序。

example.js 文件须在与 external-js.html 相同的文件夹中被创建，对应的文件夹结构如图 2.1 所示。

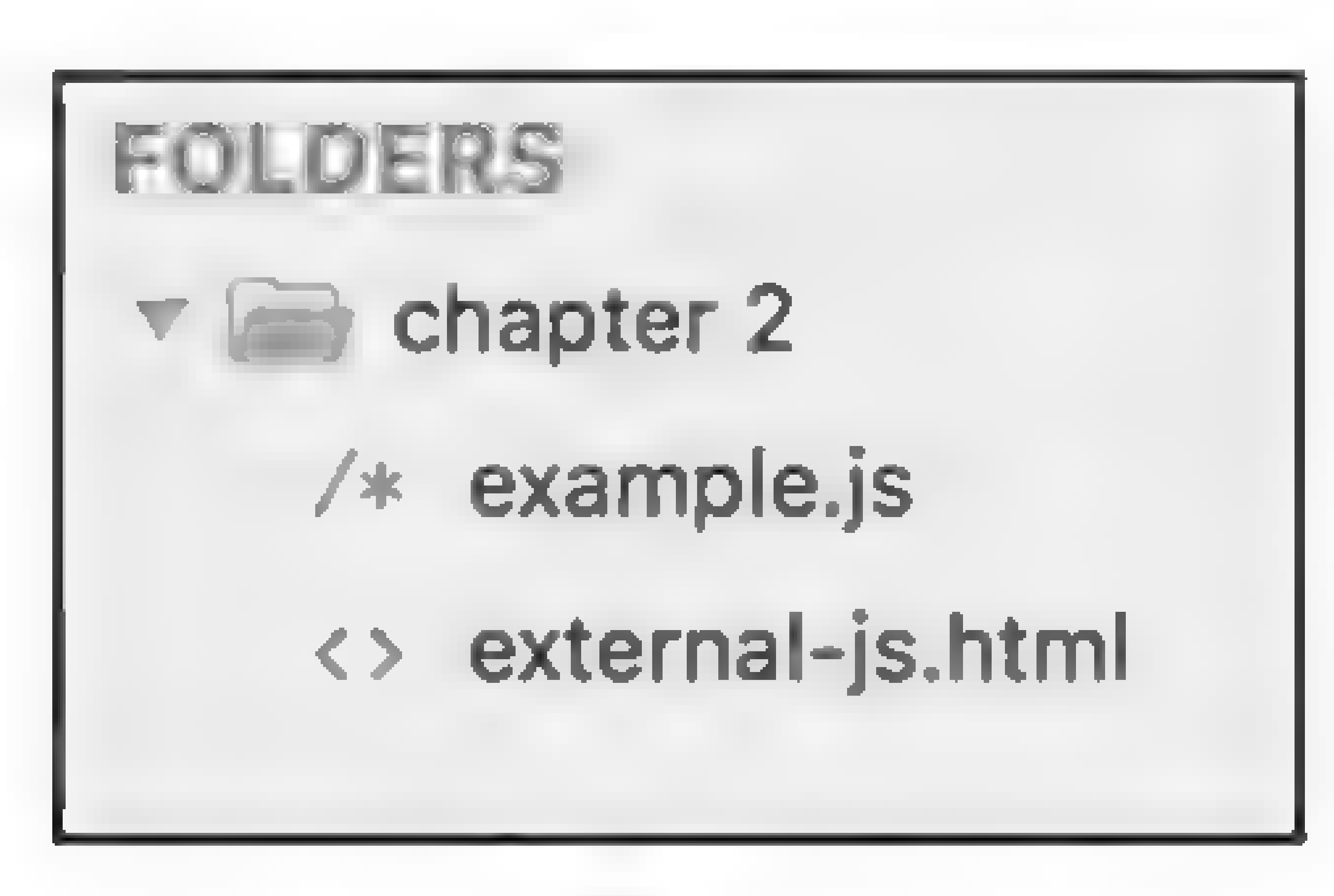



图　2.1

## 2.2　访问 JSON 中的对象

前述内容讨论了如何生成脚本调用，以获取外部 JavaScript 文件，下面采用相同的技术导入 JSON 提要。对此，我们生成了包含 100 条记录的 employee JSON 数据提要以供测试使用。在遍历 JSON 提要时，应注意数据的排列方式。数据提要中的键是基本的员工信息，如员工编号、出生日期、姓名、性别、雇用日期、职称，以及获得职称的日期。另外，少数员工在整个任职内拥有同一职务，而有些员工则拥有多个职务。

注意，这一 JSON 文件将视为代码文件的一部分内容而存在。

考查下列 JSON 存储，我们将在此基础上执行相关操作。

```
let data json = [
 {
   "emp no": "10001",
```

```
    "birth date": "1980 09 02",
    "first name": "Georgi",
    "last name": "Facello",
    "gender": "M",
    "hire date": "2010 09 04",
    "designation": {
      "title": "Junior Engineer",
      "From date": "2010 09 04",
      "to date": "2017-10-11"
    }
  },
  {
    "emp no": "10002",
    "birth date": "1970-09-02",
    "first name": "Will",
    "last name": "Scott",
    "gender": "M",
    "hire date": "2005-09-04",
    "designation": {
      "title": ["Senior Engineer", "Author", "Trainer"],
      "From date": "2005-09-04",
      "to date": "2017-10-11"
    }
  },
  {
    "emp no": "10003",
    "birth date": "1960-09-02",
    "first name": "Jenny",
    "last name": "Souza",
    "gender": "F",
    "hire date": "2006-10-05",
    "designation": {
      "title": "Architect",
      "From date": "2006-10-05",
      "to date": "2017-10-11"
    }
  }
  // And so on
]
```

考虑到将处理复杂的 JSON 数据提要，对此，需要将数据提要保存至某个文件中。在 data json feed.html 文件中，我们导入了 data.json 文件，该文件与 HTML 文件处于同一个文件夹中。

需要注意的是，JSON 提要被赋予一个名为 data_json 的变量中；当访问 JSON 提要时，需要在 data_json_feed.html 文件中使用该变量，如下所示。

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Javascript</title>
    <script type="text/javascript" src="data.js"></script>
    <script type="text/javascript">
      console.log(data_json);
    </script>
  </head>
  <body>
    <h2>Include External JSON</h2>
    <p>This is a test program to Learn How external JSON
      feed stored in files can be included.
    </p>
  </body>
</html>
```

另一个需要注意的是，console.log()新方法的使用。Mozilla Firefox、谷歌 Chrome 和苹果 Safari 等浏览器配置了一个 Console 面板，用于运行时的 JavaScript 的开发和调试。另外，JavaScript 中的 alert()函数的使用鉴于其突兀的行为而受到限制。相比之下，console.log 则不具备这种干扰性，它将消息记录到 Console 面板中。自此，我们将不再使用 alert()方法，而是使用 console.log()方法将数据输出至 Console 窗口中。谷歌 Chrome 和苹果 Safari 均配置了开发工具；当查看 Console 时，可右击 Web 页面并选择 Inspect Element 选项。另外，两个浏览器均配置了 Console 选项卡并可与日志记录协同工作。Firefox 则依赖于 Firebug，第 5 章将讨论 Firebug 的安装步骤，如图 2.2 所示。

图　2.2

当把 data_json_feed.html 文件载入至 Firefox 浏览器时，打开 Console 窗口并单击 DOM 选项卡，我们将看到一个包含 100 个 employee 对象的列表。如果对象较小且包含 1 或 2

个对象，可优先使用索引对其进行访问。在当前示例中，鉴于包含了大量的子对象，因而无法根据静态索引定位对象。对此，可查看下列示例代码。

```
/** Not realistic, unless we are targeting a specific key.**/
Ex: console.log(employees[1].emp no);
```

## 2.3　执行复杂的操作

当处理对象数组时，可通过某种迭代方法解决此类问题。对此，我们将采用一种迭代方案一次定位一个对象。当访问该对象时，则无须再次对其进行定位，从而维护数据的完整性，避免多次访问同一对象并消除冗余操作。在 JavaScript 中，循环语句表示为 while 循环和 for 循环。接下来考查如何使用循环遍历 employee 数组，对应代码如下所示。

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Javascript</title>
    <script type="text/javascript" src="data.js"></script>
    <script type="text/javascript">
    console.log(data json);
    const employeeCount = data json.length;
    for(let i=0;i<employeeCount;i++){
      console.log("Employee number is ", data json[i].emp no);
    }
    </script>
  </head>
  <body>
    <h2>Parse JSON Feed using While</h2>
    <p>This is a test program to learn how external JSON
     feed stored in files can be parsed using the for Loop.</p>
    </p>
  </body>
</html>
```

在 employees traversal.html 文件中，我们导入了 data.js 文件。data.js 文件中的 data json 变量包含了导入当前 HTML 页面中的对象数组。在<script>标签中，此处设置了两个变量。其中，变量 i 加载了一个计数器；employeeCount 变量则加载了 data json 中全部对象的数量。当检索数组中的数据项数量时，可采用 JavaScript 提供的.length 属性。for 循环中涵盖了 3 个较为重要的代码块，即条件语句、执行语句，以及基于对应条件的递增或递减语句。下面分别对其进行简要介绍。

```
let i=0;
```

这里，变量 i 初始化为 0，对应条件表示为：如果 0 小于常量 data_json 中数据项的数量，则执行循环部分。

```
i<employeeCount
```

如果条件为 true，则执行循环中的语句，直至遇到递增条件，如下所示。

```
//We are using incremental operation
i++;
//Incase of decremental operation use i--
```

一旦到达递增运算符，i 值加 1，并返回 for 循环的最初步骤。其间，将再次验证条件，并判断 i 是否小于 data_json 中数据项的数量。如果为 true，则再次进入 for 循环并执行其中的语句。该过程重复执行，直至变量 i 值等于 employeeCount。此时，for 循环的执行过程结束，for 循环中的语句将作为浏览器的 Console 窗口中的日志予以维护。在运行 HTML 文件 employees_traversal.html 之前，应确保 data.json 文件和该 HTML 文件位于同一目录。随后，将 HTML 文件加载至浏览器中（推荐使用 Chrome、Firefox 或 Safari），在 Web 页面上右击并选择 Inspect Element 选项（针对 Chrome 或 Safari 浏览器）进而打开 Console 窗口。图 2.3 中显示了 Console 窗口中的 employee 数量。

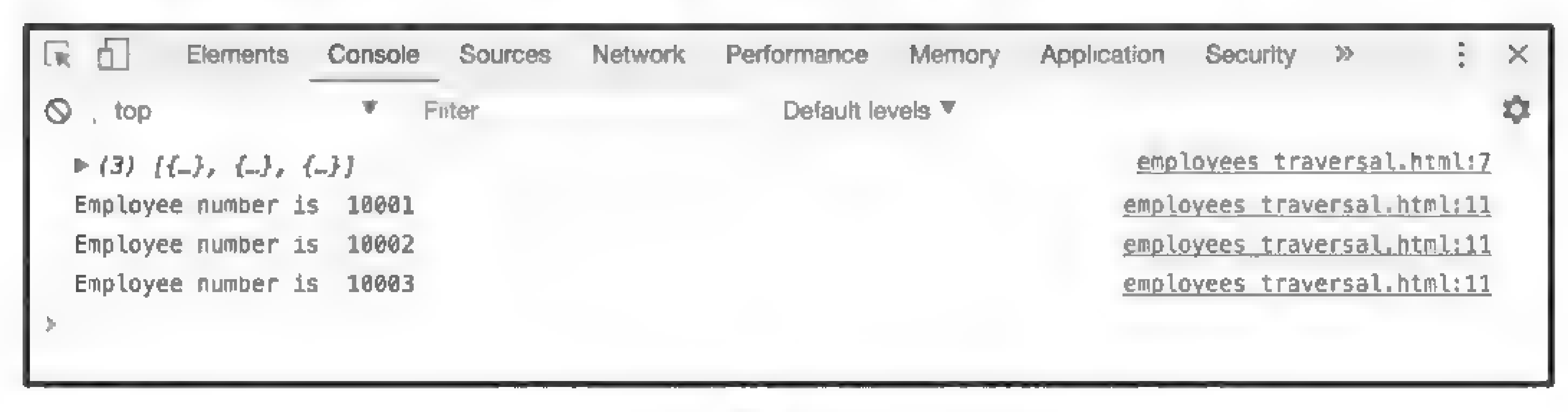



图　2.3

当检索 employee 的姓名时，可连接 employee 对象中的 first_name 和 last_name 键，如下所示。

```
//To retrieve the full name
console.log(`Employee's full Name is ${data_json[i].first_name}
${data_json[i].last_name}`);
```

除了 designation 之外，我们也可采用相同的技术检索其他键，例如 birth_date、gender 和 hire_date。快速浏览一下 JSON 提要就会发现与其他键不同，title 是对象或对象数组。designation 对象包含了员工自入职以来的所有职务。其中，一些员工仅包含一个 title；而

某些员工则包含了多个 title。前者自身即为一个对象，而后者则表示为一个对象数组，每个对象中包含了一个 title 对象。当处理此类问题时，需要检测 employee 是否包含了一个或多个 title。相应地，如果员工包含了一个 title，则可直接输出数据；否则，需要遍历 title 数组，并输出员工包含的所有 designation。对应代码如下所示。

```
for(let i=0;i<employeeCount;i++){
  console.log("Employee number is ", data_json[i].emp_no);
  console.log(`Employee's full Name is ${data_json[i].first_name}
  ${data_json[i].last_name}`);
  if(data_json[i].designation.title instanceof Array){
    const designationCount = data_json[i].designation.title.length;
    console.log("data_json", data_json[i].designation.title);
    let designations = "";
    for(let j=0;j<designationCount;j++){
      designations+= `, ${data_json[i].designation.title[j]}`;
    }
    console.log(`Employee ${data_json[i].emp_no} has served as
    ${designations}`)
  }else{
    //Employee with only one designation
    let designation = data_json[i].designation.title;
    console.log(`Employee ${data_json[i].emp_no} has served as
      ${designation}`)
  }
}
```

其中，我们使用了变量 i 和 employeeCount，同时引入了新的条件，以检测特定员工的 designation 中的 title 键是否为一个 Array 对象。此条件获取循环传递的值的类型，并验证它是否是 Array 对象的实例。下列代码将检测实例的类型。

```
if(data_json[i].designation.title instanceof Array)
```

一旦条件满足，则执行条件内的语句。在成功条件中，我们声明了 3 个变量。其中，第一个变量为 j，并针对遍历 title 的第二个 for 循环加载一个计数器；第二个变量是 designationCount，该变量存储了 title 数组中的有效数据项的数量；最后一个变量则是 designations，该变量初始化为空字符串，并加载员工拥有的全部 title，另外，该变量将存储一个 title 列表（逗号“,”分隔的字符串）。对应代码如下所示。

```
for(let j=0;j<designationCount;j++){
  designations+= `, ${data_json[i].designation.title[j]}`;
}
```

在 for 循环中，将创建员工的职务（title）信息，每次添加一个 title 至 designations 变量中。一旦 title 被成功地添加至 designations 变量中，则 j 值加 1 且循环操作持续进行，直至全部 title 字符串均被遍历；如果 title 键不是一个数组，则执行流程进入 else 代码块，并执行该块中的语句。由于对应员工仅包含一个 title，因而数据将直接输出至 Console 中。

尽管可采用 for 循环，但新的 ECMAScript 字符串修饰符可以将数组连接到默认以逗号（,）分隔的字符串中，如下所示。

```
console.log(`${data_json[i].designation.title}`)
```

上述操作类似于之前 employee 代码逻辑中的 else。“`${}`”可直接用于将数组转换为一个字符串。但需要注意的是，当采用逗号之外的连接运算符进行操作时，可能会使用到 for 循环。

至此，相信读者已对如何访问对象、执行复杂操作以析取数据有所了解，接下来将讨论如何修改 JSON 数据。

## 2.4　修改 JSON

源自 JSON 提要的 JSON 通常为只读数据。因此，JOSN 提要无法修改来自未经验证的数据源中的数据。但许多时候，我们需要从外部数据提要中获取数据，并根据需要修改其中的内容。例如，一家公司采用数据供应商提供的数据提要，但实际内容超出了公司的当前需求。此时，我们仅须析取部分内容即可，并在此基础上执行相关操作对其进行修改，并于随后复用新的 JSON 对象。下面考查 employee JSON 提要示例。

假设公司名称在不同时期发生了变化，我们需要通过公司名称并根据入职时间对员工进行分组。具体来说，2006 年之前入职的员工将被分组至 Company1 中；2006 年入职的员工将被分组至 Company2 中。当体现这一变化时，可向 JSON 提要中添加 company 键，如下所示。

```
<!DOCTYPE html>
<html>
  <head>
    <title>Modify JSON based on joining year</title>
    <script type="text/javascript" src="data.js"></script>
    <script type="text/javascript">
      for(let i in data_json){
        let data = data_json[i];
        //retrieve the year
        const join_year = parseInt(data.hire_date.slice(0,4));
```

```
      if(join_year> 2006){
        data.company = "Company1";
      }else{
        data.company = "Company2";
      }

      let message = `Employee ${data.emp_no}
        joined in the year ${join_year}
        belongs to ${data.company}`;

      console.log(message);
    }
  </script>
 </head>
 <body>
  <h2>Modifying JSON based on joining year</h2>
  <p>This is a test program to learn how JSON is imported
    from a feed could be locally modified.</p>
  </p>
 </body>
</html>
```

在 modify_employee.html 文件中，将遍历员工对象数组并获取员工的入职年份。随后将字符串转换为一个整数值，并用于公司的年份值。下列代码将解析后的年份值赋予 join_year 变量中。

```
let data = data_json[i];
//retrieve the year
const join_year = parseInt(data.hire_date.slice(0,4));
```

下列代码将检测员工的入职年份是否在 2006 年之前。若是，可将 company 属性添加至 employee 对象中，并赋予 Company1 值；否则赋予 Company2 值，如下所示。

```
if(join_year> 2006){
  data.company = "Company1";
}else{
  data.company = "Company2";
}
```

在为新添加的属性 company 赋值后，可构建一条通用消息，该消息将应用于所有员工，而不管他们属于哪个公司。这里，我们获取了生成该消息所需的员工号、员工的入职年份和格式名称，如下所示。

```
let message = `Employee ${data.emp_no}
  joined in the year ${join_year}
  belongs to ${data.company}`;
console.log(message);
```

当在浏览器中运行 modify_employee.html 时，将执行修改后的脚本，输出结果将显示于 Console 中，如图 2.4 所示。

Elements Console Sources Network Performance Memory Application »

top Filter Default levels ▼

```
Employee 10001                                         modify_employee.html:22
            joined in the year 2010
            belongs to Company1
Employee 10002                                         modify_employee.html:22
            joined in the year 2005
            belongs to Company2
Employee 10003                                         modify_employee.html:22
            joined in the year 2006
            belongs to Company2
>
```

图 2.4

## 2.5 本章小结

本章讨论了处理静态 JSON 提要的核心概念。首先是将外部 JSON 对象导入 HTML 文件中、遍历复杂的对象数组并析取所需的数据。其间，我们使用了 while 循环和 for 循环遍历数组，并通过相关条件定位搜索结果。随后，本章还对已有的 JSON 提要进行修改，并加入了新的属性。至此，相信读者已经掌握了如何从静态文件中访问 JSON，接下来将执行异步调用并通过 HTTP 获取处于活动状态下的 JSON。

# 第3章　基于 JSON 的 AJAX 请求

当前，JSON 已被视作最为流行的数据交换格式之一。在第 2 章中，我们曾考查了一个示例，并将 JSON 提要作为数据存储。本章将讨论动态数据，主要涉及以下主题。

- Web 应用程序的操作过程。
- 同步和异步请求。
- 针对 AJAX 请求设置需求条件。
- 托管 JSON。
- 采用回调、Promise 和生成器处理 AJAX 响应。
- 解析 JSON 响应。

HTML、客户端 JavaScript 和 CSS 分别提供了结构、行为和表现方面的内容；而动态 Web 开发则与客户端和服务器间的数据传输相关，我们将采用 Web 服务器、数据库和服务器端编程语言等程序获取和存储动态数据。下面将考查数据操作背后的处理过程。

## 3.1　基本的 Web 操作

当用户打开 Web 浏览器并输入 URL 时，例如 http://www.packtpub.com/，将会产生以下活动序列。

（1）浏览器请求互联网服务提供商（Internet Service Provider，ISP）通过提供域名来执行 IP 地址的反向查找。

（2）一旦获取 IP 地址，请求将被转发至拥有该 IP 地址的机器上。此时，某个 Web 服务器正在等待使用该请求。这里，Web 服务器可能是 Apache、IIS、Tomcat 或 Nginx 之一。

（3）Web 服务器接收请求并查看 HTTP 请求中的数据头。此类数据头将传递与 Web 服务器请求相关的信息。

（4）一旦 Web 服务器解析了此类数据头，将把请求路由至服务器端的应用程序中，以处理此类请求。对应的应用程序可能采用 PHP、C#/ASP.NET、Java/JSP 等语言编写。

（5）服务器端程序接收、解析请求，并执行完成请求所需的业务逻辑。这一类 HTTP 请求的相关示例包括加载一个 Web 页面和单击网站上的 Contact us 链接。当然，也存在一些较为复杂的 HTTP 请求，其间，数据需要被验证、清洗并从数据存储应用程序中被检索，例如数据库、文件服务器或缓存服务器。

HTTP 请求可通过两种方式生成，即同步请求和异步请求。

同步请求是一种阻塞请求，其间，所有事物都必须以一种有序的方式，一个接一个地完成。也就是说，下一个步骤须等到上一个步骤完成后方可执行。假设加载页面时，页面上存在 4 个独立组件。如果某个组件在执行期间占用较长的时间，那么，页面中的其他部分将处于等待状态，直至该组件执行完毕；如果执行过程中出现故障，页面加载过程也随之失败。另一个例子是 Web 页面上的投票和评分组件。在用户投票、评级后，如果采用了同步请求机制，那么这两个请求将被逐一发送。

在上述投票操作中，读者不要将其与 HTTP 长轮询技术混淆，二者是完全不同的机制。对此，读者可访问 https://en.wikipedia.org/wiki/Push_technology#Long_ polling 以了解更多信息。

当处理同步请求问题时，开发社区在异步 HTTP 请求领域中逐渐取得了进展。其中，Microsoft 首先推出了 IFrame 标签，进而使用 IFrame 并通过 IE 浏览器异步加载内容。在 IFrame 之后则是 XML HTTP ActiveX 控件。后来，所有浏览器都在新的 XMLHttpRequest JavaScript 对象（XMLHttpRequest API 中的一部分内容）下采用这种控件。XMLHttpRequest API 用于向 Web 服务器生成 HTTP（或 HTTPS）调用，并可实现同步和异步调用。相应地，异步请求允许开发人员将 Web 页面划分为多个彼此无关的组件，并根据需要发送数据，从而节省了大量的内存空间。

Jesse James Garrett 将这一现象称作 AJAX（即异步 JavaScript 和 XML），Web 请求通过 JavaScript 所生成，数据交换最初产生于 XML 中。AJAX 中的 X 当初被视为 XML，但当今它可以是任何数据交换格式，例如 XML、JSON、文本文件，甚至是 HTML。用于数据传输的数据格式须呈现于 MIME 类型数据头中。在第 1 章中，我们特意强调了 JSON 成为首选数据交换格式的原因。接下来将利用 JSON 数据生成第一个 AJAX 调用。

从本质上讲，Web 开发人员可以使用 AJAX 原则按需获取数据，从而使网站更具响应性和交互性。当然，理解这种需求的根源是非常重要的。相应地，这种数据需求的触发器通常是发生在 Web 页面上的事件。这里，事件可描述为对所执行动作的反应。例如，摇铃这一动作将在铃铛内生成振动，进而产生声音。因此，可将摇铃这一动作则视为当前事件，而产生的声音则被视为对事件的响应。一个 Web 页面上可存在多个事件，例如单击按钮、提交表单、鼠标指针悬停于某个链接上、从下拉菜单中选取某个选项，这都是十分常见的事件。当出现此类事件时，我们必须找到一种以编程方式处理它们的方法。

## 3.2　AJAX 需求

AJAX 是浏览器（客户机）和活动 Web 服务器之间通过 HTTP（或 HTTPS）的异步双向通信。其中，可通过本地方式并使用 Apache、IIS 或 node.js 这一类工具运行活动服

务器。这里，我们将选择 node.js 作为主服务器平台。

node.js 服务器的搭建过程涉及以下步骤。

（1）访问 https://nodejs.org/en/并下载安装文件。用户可选择 LTS (long-term support)可执行文件，以获取 node.js 社区的长期支持。

（2）安装过程为简单的一键式处理，用户遵循相关步骤即可。如果用户打算从终端进行安装，则可单击链接 https://nodejs.org/en/download/package-manager/。

（3）安装完毕后，即可设置并编写第一个服务器端程序，进而生成/托管一个 JSON 提要。在终端中运行以下命令将在当前目录下生成一个文件，如下所示。

```
$ mkdir test-node-app
$ cd test-node-app
$ npm init
```

npm init 需要在设置应用程序之前完成某些附加信息，该命令对于创建 package.json 十分重要，同时担任了所有安装模块和数据包的注册器。此外，它还在管理应用程序时扮演了主要的角色。读者可访问 https://docs.npmjs.com/cli/init 以了解更多信息。

（4）生成名为 app.js 的应用程序，该文件包含了构建服务器的代码，如下所示。

```
const http = require('http');
const port = 3300;
http.createServer((req, res) => {
  res.writeHead(200, {
    "Content-Type": "text/plain"
  });
  res.write("Hello Readers!");
  res.end();
}).listen(port);
console.log(`Node Server is running on port: ${port}`)
```

（5）运行下列命令。

```
node app.js
```

如果一切运转正常，将得到下列输出结果。

```
Node Server is running on port: 3300
```

当执行服务器请求操作时，须打开操作系统中的浏览器，并访问 http://localhost:3300。随后，在浏览器文档主体中将得到下列输出结果。

```
Hello Readers!
```

一旦接收到这条消息，即可确信 Web 服务器已经启动并处于运行状态。

下面将逐步分析上述脚本的工作方式。

（1）第一行代码包含节点 API 中已存在的 HTTP 本地模块；该模块提供了构建 HTTP 服务器所需的全部功能。

（2）针对所提供的各项功能，HTTP 实例包含了多种方法。此处我们使用了 createServer()方法。当调用该方法时，将创建一个服务器并返回一个 HTTP 服务器实例。

（3）可作为参数向 createServer()方法传递一个回调函数，这样每当服务器接收到一个 HTTP 请求时，该方法就会被调用并向客户端发送所需的响应。虽然作为参数传递的回调函数是可选的，但对接收到的请求执行各种操作并返回相应结果则是十分重要的。

（4）最后，最重要也是强制性的步骤是监听指定端口。这可以通过调用 HTTP 服务器实例提供的 listen()方法来实现。

（5）至此，HTTP 服务器已准备就绪，但会采用与"Hello Readers! "不同的字符串数据格式响应客户端，因而需要转换为 JSON。

Web 应用程序可构建于任何语言之上；同时，在服务器端堆栈所支持的 Web 应用程序之间，JSON 可用作相应的数据交换语言。接下来将讨论服务器端编程语言，并在 Web 应用程序对其予以实现。

## 3.3　托管 JSON

本节将创建一个节点脚本，该脚本允许我们在成功请求时向用户发送 JSON 反馈。下列代码显示了实现这一任务的 app.js 文件。

```
const http = require('http');
const port = 3300;
http.createServer((req, res) => {
  res.writeHead(200, {
    "Content-Type": "application/json"
  });
  res.write(JSON.stringify({
    greet: "Hello Readers!"
  }));
  res.end();
}).listen(port);
console.log(`Node Server is running on port: ${port}`)
```

上述代码突出显示了发送 JSON 数据时所做的修改内容。该脚本由 JSON 对象构成，其中，greet 作为键，Hello Readers!表示为值。由于所提供的响应结果总是以字符串或缓

冲区格式呈现，因而对象首先被字符串化。除此之外，还需要将内容类型提供为 application/json。Content-Type 将在响应头中进行设置，以便浏览器可以识别所提到的响应类型。

下面在 student 数据的基础上进一步丰富 JSON 内容。下列代码创建了一个名为 student_data.json 的 JSON 文件。

```
//student_data.json
[{
  "studentid": 101,
  "firstname": "John",
  "lastname": "Doe",
  "classes": ["Business Research", "Economics", "Finance"]
},
{
  "studentid": 102,
  "firstname": "Jane",
  "lastname": "Dane",
  "classes": ["Marketing", "Economics", "Finance"]
}]
```

服务器脚本文件中的 student_data.json 文件如下所示。

```
const http = require('http');
const port = 3300;
const studentsData = require('./student_data.json');
http.createServer((req, res) => {
  res.writeHead(200, {
    "Content-Type": "application/json"
  });
  res.write(JSON.stringify(studentsData));
  res.end();
}).listen(port);
console.log(`Node Server is running on port: ${port}`)
```

上述节点脚本生成了一个 studentsData 数组，同时针对该数组生成了一个 JSON 提要。studentsData 数组中包含了基本的学生信息，例如名字、姓氏、学生 ID 以及学生所注册的班级。

接下来通过节点 Web 服务器访问该文件。在访问 http://localhost:3300 后将得到如图 3.1 所示的结果。

```
[{"studentid":101,"firstname":"John","lastname":"Doe","classes":["Business
Research","Economics","Finance"]},
{"studentid":102,"firstname":"Jane","lastname":"Dane","classes":
["Marketing","Economics","Finance"]}]
```

图　3.1

当通过节点 Web 服务器运行该文件时，服务器将接收请求信息并对其进行处理，随后输出传递学生数据的 JSON 提要。

## 3.4　第一个 AJAX 调用

前述内容创建了一个 JSON 数据提要，本节将生成第一个 AJAX 调用。对此，我们将考查不同的方案，这项技术及其使用方式已经发展了一段时间，旨在提高性能。第一种方案将采用基本的 JavaScript 方法，以便理解 AJAX 调用背后的流程；随后我们将使用一些较为流行的库，在稍作调整后生成相同的 AJAX 调用。在第一种方案中，将通过下列模板和 JavaScript 创建 basic.html 文件。

```
<!DOCTYPE html>
<html>
<head>
  <title>First AJAX script</title>
  <script type="text/javascript" src="basic.js"></script>
</head>
<body>
  <h2>Include external Javascript to make an ajax call</h2>
  <p>This is a test program to make our first AJAX call using
  javascript</p>
</body>
</html>
```

此处将从加载外部 JavaScript 文件的 basic.html 文件开始，此类 JavaScript 文件将执行 AJAX 调用以获取 students JSON 提要。

考查下列 basic.js 文件。

```
const request = new XMLHttpRequest();
request.open('GET', 'http://localhost:3300');
request.onreadystatechange = function(){
  if((request.status==200) && (request.readyState==4)){
    console.log(request.responseText);
  }
}
request.send();
```

这也是对 Web 服务器进行 AJAX 调用的原始方式。下面将上述脚本分为几部分内容并逐一进行讨论。

```
const request = new XMLHttpRequest();
```

上述代码片段生成了一个 XMLHttpRequest 对象实例。XMLHttpRequest 对象可向服务器进行异步调用，并将页面中的内容视为独立的组件；其中包含了 readystate、response 和 responseText 等有用的属性，以及 open、onuploadprogress、onreadystatechange 和 send 等方法。下面查看如何使用生成的 request 对象打开一个 AJAX 请求，如下所示。

```
request.open('GET', 'http://localhost:3300');
```

默认状态下，XMLHttpRequest 将打开一个异步请求。此处，我们将指定获取提要所需的对应方法。鉴于当前不会传递任何数据，因而可选择 HTTP GET 方法，并将数据发送至 Web 服务器上。当在异步请求上进行操作时，不应设置阻塞脚本，可通过设置回调对此加以处理。这里，回调是指一组等待响应的脚本，并在接收响应结果时被触发。这种行为有助于非阻塞代码。

下面设置一个回调，并将其赋予 onreadystatechange 方法中，如下所示。

```
request.onreadystatechange = function(){
  if((request.status==200) && (request.readyState==4)){
    console.log(request.responseText);
  }
}
```

占位符方法 onreadystatechange 在请求对象中查找一个名为 readyState 的属性；当 readyState 值发生变化时，即会触发 onreadystatechange 事件。readyState 属性负责跟踪 XMLHttpRequest 的进程。在图 3.1 中，可以看到回调包含了一个条件语句，用于验证 readyState 的值为 4。这意味着，服务器已接收到客户端生成的 XMLHttpRequest，同时响应已经就绪。

表 3.1 显示了 readyState 的有效值。

表　3.1

| readyState | 描　　述 |
|---|---|
| 0 | 请求尚未被初始化 |
| 1 | 服务器连接已建立 |
| 2 | 服务器已接收到请求 |
| 3 | 服务器正在处理请求 |
| 4 | 请求已被处理完毕，响应已就绪 |

在之前的代码中，我们曾查询过一个属性 status，这表示为从服务器返回的 HTTP 状态代码。具体来说，状态代码 200 表示一个成功的事务；而状态代码 400 则表示一个糟糕的请求；404 表示没有找到页面。其他一些较为常见的状态码还包括 401（表示用户请求了一个仅针对授权用户的页面）和 500（表示内部服务错误）。

前述操作曾创建了一个 XMLHttpRequest 对象并打开了连接；此外，还添加了一个回调，并在请求成功后执行某个事件。需要注意的是，该请求尚未生成，我们只是完成了该请求所涉及的基础工作。随后，将使用 send()方法将请求发送至服务器，如下所示。

```
request.send();
```

在 onreadystateChange 回调中，我们将 Web 服务器发送的响应输出至控制台窗口中。当首次运行代码时，浏览器的 Console 选项卡将会显示如图 3.2 所示的内容。

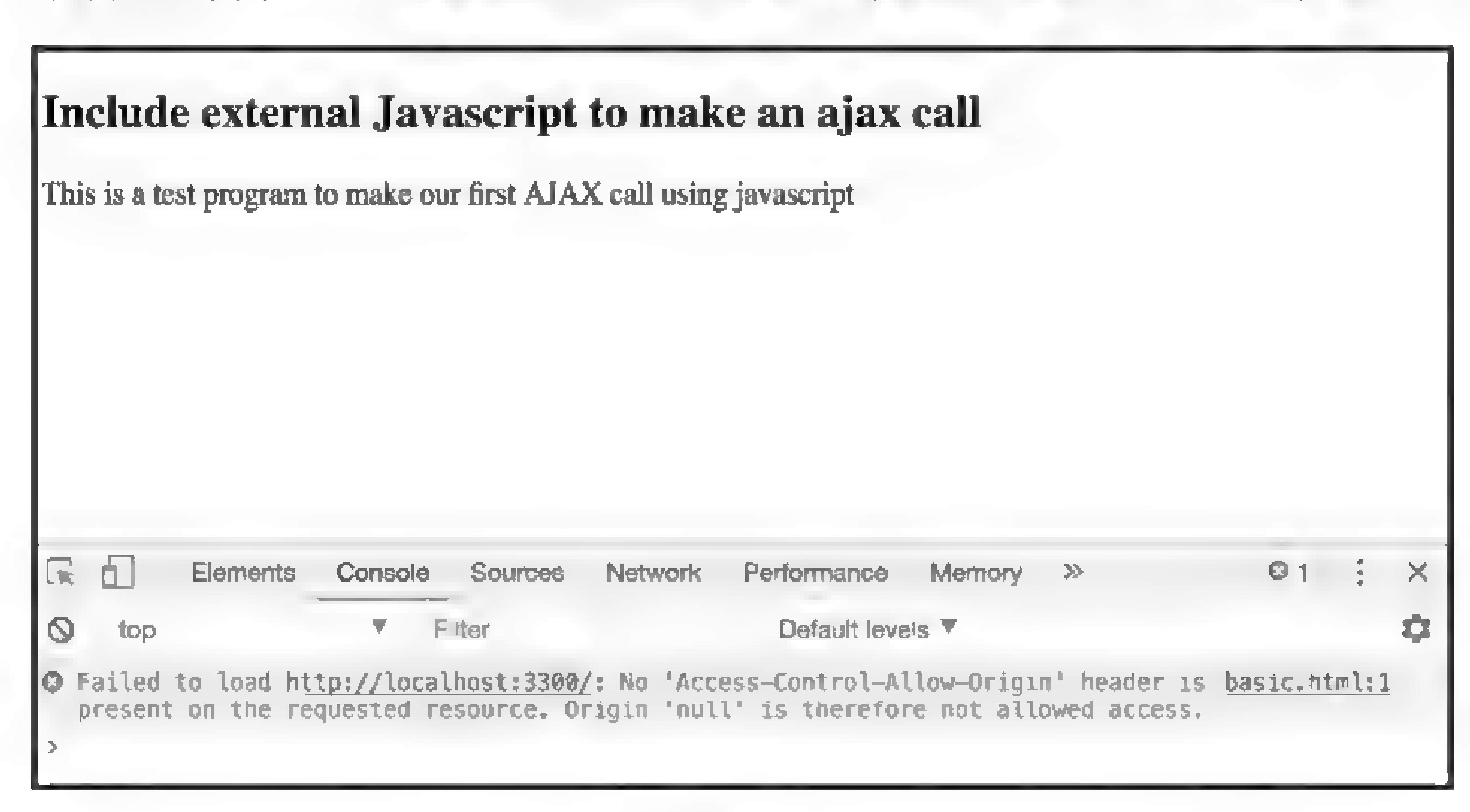



图　3.2

错误的原因在于，加载的文档在浏览器的 URL 定位器中包含不同的域名，而 AJAX 则请求一个不同的 URL，这源自浏览器的跨域资源共享（Cross-Origin Resource Sharing，CORS）策略。关于 CORS，读者可访问 https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS 以了解更多内容。

针对这一问题，我们需要在服务器端对其加以处理，以提供 CORS 方面的支持。对此，须在 writeHead() 方法或 res 响应对象中添加 Access-Control-Allow-Origin。对应值为"*"，表示访问被授予任何域或请求，如下所示。

```
res.writeHead(200, {
  "Content-Type": "application/json",
  "Access-Control-Allow-Origin": "*",
});
```

这里，应确保设置内容不同于生产环境。在生产环境中，作为示例，域名可指定为下列内容。

```
"Access-Control-Allow-Origin":"www.differentdomain.com"
```

此时，响应的输出结果如图 3.3 所示。

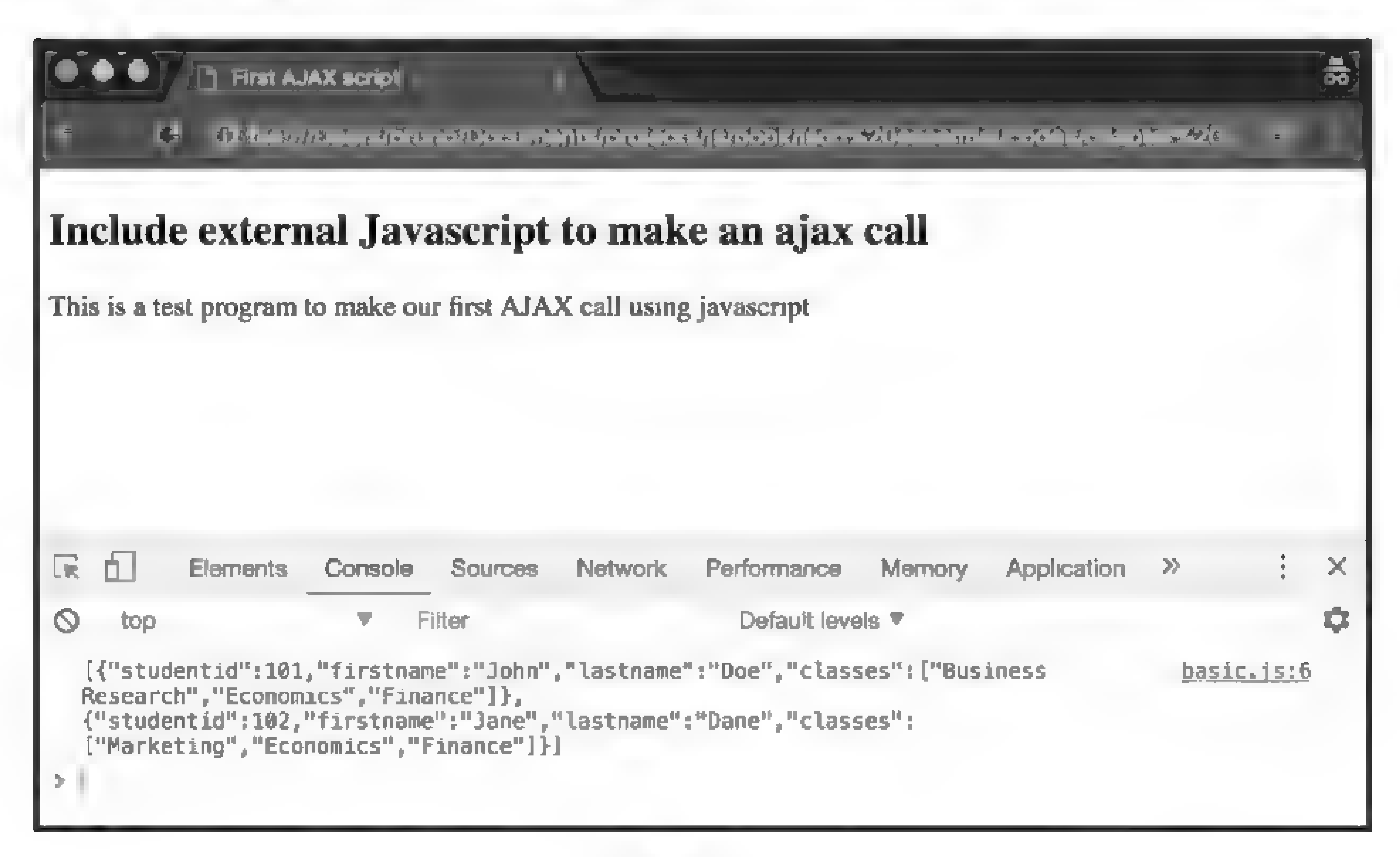



图　3.3

一种确认 AJAX 请求的方式是，查看浏览器的 Network 选项卡（位于 Console 选项卡附近），其中对 localhost:3300 地址进行异步调用，返回的响应包含了 200 OK 的 HTTP 状

态码。由于 HTTP 状态码为 200，这表明回调成功执行，同时输出 students JSON 提要。

随着强大的 JavaScript 库（例如 jQuery、Scriptaculous、Dojo 和 ExtJS）的出现，AJAX 请求已较少出现。需要注意的是，库仍然会在底层使用这一过程。因此，了解 XMLHttpRequest 对象的工作方式非常重要。jQuery 是一个非常流行的 JavaScript 库，并拥有一个不断壮大的社区。由于 jQuery 库是在 MIT 许可下发布的，所以用户可以免费使用这个库。

jQuery 是一个非常简单、功能强大的库，具有出色的文档和强大的用户社区，这使得开发人员的工作非常简单。下列代码采用 jQuery 实现了 Hello World 程序。

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello world using jQuery</title>
    <!-- Go to https://code.jquery.com/ for more details -->
    <script src="https://code.jquery.com/jquery-3.2.1.min.js"
    integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
    crossorigin="anonymous"></script>
    <script>
    $(document).ready(() => {
       console.log("Hello world!");
    })
    </script>
  </head>
  <body>
    <h2>Hello world using jquery</h2>
    <p>This is a Hello world program using jquery</p>
  </body>
</html>
```

上述代码向 HTML 文件中导入了 jQuery 库。在第二组<script>标签中，我们使用了特定的字符$或 jQuery。与面向对象程序设计中的命名空间相似，jQuery 功能的名称空间默认为特殊字符$。在$之后，这里调用了 document 对象并检测是否被加载至页面上；随后则分配了一个回调函数，该函数将在文档加载事件完毕后被触发。此处，document 表示为一个 document 对象，其中加载了 HTML 元素结构。当前程序的输出结果为 Hello World!字符串，并输出至控制台窗口中。

前述内容讨论了客户端请求的较为传统的 JSON 处理方式，并简要地介绍了回调这一概念，即处理服务器请求的异步方式之一。接下来将进一步深入讨论回调，同时介绍一些更先进的方法。

### 3.4.1　传统的回调

回调是一个简单的匿名函数，它作为参数被传递，并在需要时在另一个函数的作用域内被调用。考查下列代码片段。

```
function greetAll(callback){
  console.log("Hello Readers!");
  console.log(`Greeting from the ${callback()}`)
}
greetAll(()=>{
  return "Author";
})
```

上述代码片段中设置了一个 greetAll()函数，其中包含了一个简单的 callback 参数。当调用 greetAll()函数时，将作为参数传递一个匿名函数。此后，greetAll()函数在编程上下文中被执行，该上下文将执行以下逐项操作。

（1）首先输出 Hello Readers!。

（2）当从左至右执行第二个输出项时，将依次调用名为 callback 的参数来执行操作，因而将输出 Greeting from the Author。

注意，回调可在 greetAll()函数的全部操作执行完毕后被调用；或者，回调也可在其他操作之前被调用，这取决于所需的相关功能。这一类机制可用于处理异步事件。考查地址 localhost:3300 产生的请求，这是一个利用浏览器中 AJAX 方法所生成的异步请求。在此基础上，下面采用 jQuery AJAX GET 方法对 AJAX 调用应用回调，如下所示。

```
<!--jquery-ajax.html-->
<!DOCTYPE html>
<html>
<head>
  <title>Ajax using jquery</title>
  <!-- Go to https://code.jquery.com/ for more details -->
  <script src="https://code.jquery.com/jquery-3.2.1.min.js"
  integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
  crossorigin="anonymous"></script>
  <script>
  $(document).ready(() => {
    $('#getFeed').click(()=>{
      $.getJSON('http://localhost:3300/',(jsonData)=>{
        console.log("jsonData", jsonData);
      })
    })
  })
```

```
  </script>
</head>
<body>
  <h2>AJAX using jquery</h2>
  <input type="button" id="getFeed" value="Get Feed" />
</body>
</html>
```

在上述<script>标签中，我们向节点服务器请求获取学生数据。在当前示例中，我们将回调作为第二个参数传递给 jQuery 的$.getJSON()方法调用，该回调处理在请求完成时接收的数据。类似地，如果仔细观察，还会发现前面的代码片段中还使用了一种回调机制，即 jQuery 的 click()方法，它接受一个回调函数，该回调函数提供了一种机制，用于在浏览器中单击事件发生时编写代码。

至此，相信读者已经了解了回调的整体概念及其被应用原因。接下来将讨论一些较为高级的异步数据处理机制。

### 3.4.2　利用 Promise 处理异步操作

虽然回调应用广泛，通常也是执行异步操作的最佳方式，但它仍然无法满足某些代码模式的标准要求，如下所示。

- ❑ 如果在回调中返回数据，并传递至某个库（如 jQuery），那么将很难了解到回调中的返回数据在库代码中的处理方式。考查下列代码。

```
$.getJSON('http://localhost:3300/',(jsonData)=>{
  console.log("jsonData", jsonData);
  return jsonData;
})
```

其中，返回语句的去向尚不明晰。

- ❑ 另一个需要处理的问题是错误异常。考查下列代码。

```
$.getJSON('http://localhost:3300/',(jsonData)=>{
  console.log("jsonData", jsonData);
  throws "someError 500";
})
```

如果出现意外错误或显式地抛出异常，则无法在回调级别上处理该错误。尽管存在 try{}catch(e){}块可对此予以处理，但不建议每次在回调中编写代码时对其加以使用。

- ❑ 最后一个问题则是代码的可读性。连续使用过多的回调会导致“回调地狱”问题。对此，读者可访问 http://callbackhell.com/以了解更多内容。

针对此类问题，这里引入了 Promise 这一概念，下面先通过代码形式对其予以实现。对此，需要利用'thenable'链替换之前传统的回调方式，如下所示。

```
$.getJSON('http://localhost:3300/')
 .then((jsonData)=>{
  console.log("jsonData", jsonData);
  return jsonData;
})
.then((jsonDataAgain)=>{
 console.log("recieved jsonData again", jsonDataAgain);
 return jsonDataAgain;
})
```

Promise 背后包含了两个简单的状态，如下所示。

- resolved：表示为 Promise 的一种状态，其间，相关功能已成功完成。上述示例请求了 localhost:3300，并作为 jsonData 成功地接收到响应（响应头中的 status-code 200）。随后，该响应在 then()方法的回调中被接收。
- rejected：表示为 Promise 的一种状态，在这种状态下，相关功能无法安装予以完成。对此，下列代码在 then 的回调中抛出一个错误。

```
$.getJSON('http://localhost:3300/')
.then((jsonData)=>{
 throw "Error: Something is wrong";
 return jsonData;
})
.then((jsonDataAgain)=>{
 console.log("recieved jsonData again", jsonDataAgain);
 return jsonDataAgain;
})
.catch((error)=>{
 console.log("Error handled", error);
})
```

其中，显式地抛出错误可通过 Promise 的 catch()方法予以处理。在发送请求时或接收请求之后可能发生的所有错误都可以在 catch()方法中捕获。

注意，在第一个 then()方法的回调中所返回的任何内容都可以被同级的相邻 then()方法接收。

### 3.4.3　新的 ECMAScript 生成器

如果读者厌倦了 JavaScript 中的回调风格，或者更趋向于阻塞或暂停异步执行，并在

接收到数据时对其予以恢复，则会偏向于 ECMAScript 在 JavaScript: generator 中引入的新功能。

简单地讲，生成器是一类迭代函数，主要包括以下内容。

- 生成某种类型的数据。
- 使用 yield 语句暂停或恢复执行。
- 在语法上，通过 function *functionName 或 function*functionName 加以表示。
- 如果需要连续生成所监听的事件，可采用无限循环。

上述内容也表示为生成器的规范。为了简化定义，下面考查如何在 AJAX 示例中实现这一概念。

在该示例中，异步行表示为$.getJSON('http://localhost:3300/')，此处将在生成器函数中对其予以隔离。当编写生成器函数时，需要遵循以下指导原则。

- 包含生成器语法，如下所示。

```
function* generateData(){}
```

- 包含 yield 语句，如下所示。

```
function* generateData(){
  yield $.getJSON('http://localhost:3300/');
}
```

- 一旦生成器函数就绪，即可像其他函数那样对其加以调用，如下所示。

```
const generatorInst = generateData();
```

这里，generateData()函数将返回一个迭代器对象，而不是响应数据。

- 当获取响应数据时，若存在一个迭代器对象，则需要调用 next()方法。下列代码调用了单击按钮操作上的 next()方法。

```
$('#getFeed').click(()=>{
  console.log(generatorInst.next());
})
```

首次单击按钮后，对应输出结果如图 3.4 所示。

- 一旦获取数据并第二次单击按钮，输出结果如图 3.5 所示。

可以看到，done 的键值为 true，value 键为 undefined。这意味着，执行过程到达了生成器函数的结尾（不存在任何迭代可获取数据）。因此，自第二次单击操作开始，我们将得到类似的数据。

```
Elements  Console  Sources  Network  Performance  Memory  Application  »
top  Filter  Default levels ▼
                                                        jquery-ajax.html:22
▼{value: {…}, done: false}
   done: false
  ▼value:
   ▶abort: ƒ (a)
   ▶always: ƒ ()
   ▶catch: ƒ (a)
   ▶done: ƒ ()
   ▶fail: ƒ ()
   ▶getAllResponseHeaders: ƒ ()
   ▶getResponseHeader: ƒ (a)
   ▶overrideMimeType: ƒ (a)
   ▶pipe: ƒ ()
   ▶progress: ƒ ()
   ▶promise: ƒ (a)
    readyState: 4
   ▶responseJSON: (2) [{…}, {…}]
    responseText: "[{"studentid":101,"firstname":"John","lastname":"Doe","classes":["Business Res
   ▶setRequestHeader: ƒ (a,b)
   ▶state: ƒ ()
```

图　3.4

```
Elements  Console  Sources  Network  Performance  Memory  Application  »
top  Filter  Default levels ▼
▶{value: {…}, done: false}                              jquery-ajax.html:22
▶{value: undefined, done: true}                         jquery-ajax.html:22
>
```

图　3.5

当在所有单击行为上生成数据时，可在生成器函数 generateData()中做出如下调整。

```
function* generateData(){
  while(true){
    yield $.getJSON('http://localhost:3300/');
  }
}
```

当前，每次单击 Get Feed 按钮时，将会看到获取的响应数据。

综上所述，我们已经通过回调、Promise 和生成器等方式生成了 AJAX 请求。其中，每种机制包含了自身的优势，以及相应的 ECMAScript 版本实现。在学习了浏览器中的 JSON 获取机制之后，接下来将解析所接收的数据，以供其他操作使用。

## 3.5　解析 JSON 数据

在介绍了 jQuery 之后，下面将在某个事件上触发一个 AJAX 请求，例如单击按钮，如下所示。

```
<!--jquery-ajax.html-->
<!DOCTYPE html>
<html>
<head>
  <title>Ajax using jquery</title>
  <!-- Go to https://code.jquery.com/ for more details -->
  <script src="https://code.jquery.com/jquery-3.2.1.min.js"
  integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
  crossorigin="anonymous"></script>
  <script>
  $(document).ready(() => {
    $('#getFeed').click(()=>{
      $.getJSON('http://localhost:3300/',(jsonData)=>{
        if(jsonData){
          $.each(jsonData, (key, value)=>{
            $('#feedContainerList')
            .append(`<li>Student Id is ${value.studentid}
              and the student name is ${value.firstname}
              and ${value.lastname}
              </li>`);
          })
        }
      })
    })
  })
  </script>
</head>
<body>
  <h2>AJAX using jquery</h2>
  <input type="button" id="getFeed" value="Get Feed" />
  <div id="feedContainer">
    <div id="feedContainerList">

    </div>
  </div>
</body>
</html>
```

在上述代码片段中，让我们从查看 HTML document 对象开始。这里，div 元素包含了一个空的无序列表。这个脚本的目的是利用单击按钮时的列表项填充无序列表。input 按钮元素的 id 值为"getFeed"，单击事件处理程序将绑定到该按钮上。考虑到 AJAX 的异步特性，同时我们为这个按钮分配了一个回调，所以在加载 document 对象时，不会对活动服务器进行 AJAX 调用，且仅将 HTML 结构加载到页面上，并将事件绑定到这些元素之上。

当单击按钮时，将通过 getJSON 方法向 Web 服务器产生 AJAX 调用，并检索 JSON 数据。考虑到将得到一个 Student 数组，因而可将检索到的数据传递至 jQuery 的 each 迭代器中，且每次检索一个元素。在该迭代器内部，将构建一个字符串，该字符串作为列表项附加到"feedContainerList"无序列表元素中，如图 3.6 所示。

**AJAX using jquery**

Get Feed

- Student Id is 101 and the student name is John and Doe
- Student Id is 102 and the student name is Jane and Dane

图　3.6

除非单击按钮，否则不会有任何行为上的改变。单击按钮后，无序列表将被填充。

## 3.6　本 章 小 结

XMLHttpRequest 对象的流行，对于 Web 开发人员来说无疑是一条好消息。本章首先讨论了这方面的基础知识，例如产生 AJAX 请求所需的相关内容。此外，本章还进一步介绍了 XMLHttpRequest 对象如何产生异步请求。接下来我们还学习了功能强大的 JavaScript 库 jQuery，并通过 jQuery 执行 AJAX 操作——这也仅是 AJAX 之旅的开始阶段。第 4 章将探讨 AJAX 更加复杂的应用场合、跨域异步请求失败，以及 JSON 如何通过跨域异步调用来节省操作时间。

# 第 4 章 跨域异步请求

第 3 章使用了 jQuery 的 getJSON 获取学生的 JSON 提要。本章将对此进一步讨论并向服务器发送请求参数。另外，第 3 章中还介绍了跨域请求，本章则在此基础上深入讲解这一话题。本章主要涉及以下内容。

- 使用 JSON 数据生成 GET 和 POST AJAX 调用。
- 与跨域调用相关的问题。
- JSONP 简介。
- JSONP 实现。

让我们从理解 API 的概念开始本章内容。

## 4.1 API

数据提要一般包含较大的数据量且兼具通用性，但这对于目标搜索来说可能过于繁重。例如，在学生 JSON 提要中，我们公开了学生信息的整个列表。如果数据供应商正在查找注册了某些课程的学生，或者居住在给定邮编内的实习生，那么，此类提要须具备通用性。通常可以看到，开发团队负责构建应用程序编程接口（Application Programming Interface，API），进而为数据供应商提供多种搜索方式。这对于数据供应商和持有信息的公司来说是一个双赢的局面，因为数据供应商只获得他们想要的信息，而数据提供商只发送请求的数据，因此节省了大量的带宽和服务器资源。

## 4.2 利用 JSON 数据生成 GET 和 POST 调用

需要注意的是，我们应理解同步和异步调用都是通过 HTTP 进行的，因此数据传输过程是相同的。对于客户端至服务器间的数据传输，较为流行的方法是 GET 和 POST；而最为常见的请求方法则是 GET。当客户端请求一个 Web 页面时，Web 服务器使用 URL 处理 HTTP 请求。相应地，附加到 URL 的任何其他参数都用作从客户机发送到服务器的数据。鉴于参数也是 URL 中的一部分内容，因而明确区分何时使用 GET 请求方法（以及何时不使用 GET 请求方法）是非常重要的。GET 方法一般用于传递诸如页码、链接地址或分页中的限制条件和偏移量等信息。需要注意的是，对于 GET 请求方法可以传输的数据量，一般存在一定的限制。

我们将与一个修改后的学生 API 协同工作，并于其中查询完整的学生信息——学生所居住的邮政编码以及注册的课程；还可通过组合搜索获取居住在特定区域，同时选取了某一门课程的学生。

读者可访问以下 GitHub 链接以获取本章的示例代码。

```
https://github.com/bron10/jsonessentials-book/tree/master/
  chapter%204
```

下列内容列出了第一个目标搜索（通过邮编）的 URL。

```
http://localhost:3300/?zipcode=400002
```

该 API 调用将返回居住在给定邮编区域内的全部学生信息，对应输出结果如图 4.1 所示。




图　4.1

在上述示例中，我们接收到一名学生的详细信息，对应的邮政编码为 400002。针对这一输出结果，需要在节点服务器 app.js 文件中进行适当调整，如下所示。

```
const http = require('http');
const port = 3300;
const urlObject = require('url');
const querystring = require('querystring');
let studentsData = require('./student data.json');
http.createServer((req, res) => {
  let url = req.url;
  const urlParsedObject = urlObject.parse(req.url);
  const pathname = urlParsedObject.pathname;
  const queryObject = querystring.parse(urlParsedObject.query);
  res.writeHead(200, {
    "Content-Type": "application/json",
    "Access-Control-Allow-Origin": "*"
  });
  switch(pathname){
    case '/':
    let student = studentsData.filter((student)=>{
      return student.zipcode == queryObject.zipcode;
```

```
    })
    res.write(JSON.stringify(student));
    res.end();
    break;
  }
}).listen(port);
console.log(`Node Server is running on port: ${port}`)
```

首先，代码中添加了两个本地节点 require('url')和 require('querystring')。前者提供了从 URL 字符串中解析数据的方法。当前示例中包含了一个请求 URL，用于简单地解析 YRL 查询。因此，这里将?zipcode=400002 转换为 JSON 对象{zipcode: 400002}以便于访问。

关于 querystring 包的更多信息，读者可访问 https://nodejs.org/docs/latest-v7.x/api/querystring.html。

除此之外，关于 URL 包的更多信息，读者可访问 https://nodejs.org/docs/latest-v7.x/api/url.html。

需要注意的是，一旦得到了查询数据，即可遍历 studentsData 数组，并利用邮政编码键过滤学生数据。最终结果将写入响应中；在调用了响应的 end()方法后，数据将发送至客户端。

另一个需要注意的是 switch-case 语句，该语句用于区分所有路由。这样，只须在 switch 中轻松地插入一个 case，即可添加更多的路由。

get-students.html 中的代码如下所示。

```
<!DOCTYPE html>
<html>
<head>
  <title>Get all students</title>
  <!-- Go to https://code.jquery.com/ for more details -->
  <script src="https://code.jquery.com/jquery-3.2.1.min.js"
    integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
    crossorigin="anonymous"></script>
  <script>
  $(document).ready(() => {
    $.ajax({
      "url": "http://localhost:3300",
      "type": "GET",
      "data": {},
      "dataType":"JSON"
    })
    .done((data)=>{
      console.log(data);
```

```
    })
  })
  </script>
</head>
<body>
  <h2>Get all students</h2>
  <p>Retrieve the students information of all students</p>
</body>
</html>
```

代码中首先导入了 jQuery 库。由于页面中引入了 jQuery，因而可开始使用$变量。代码中添加了一个回调，并在文档就绪后被触发。考虑到将要生成 GET 和 POST 请求，当前示例使用了 ajax()方法。

此处无须显式地提及 GET 类型，但这有助于保持与代码的一致性。

在 AJAX 调用中，我们向 API 调用设置了链接的 URL 属性，以便检索学生信息，并指定这将通过 HTTP GET 方法予以执行。代码中设置的第 4 个属性是 dataType，表示期望返回的数据类型。由于与学生的提要信息协同工作，因而需要将 dataType 属性设置为 JSON。这里应注意，服务器向异步请求发送响应时所触发的 done 回调。我们将从服务器发送来的数据作为响应予以传递，并启动回调。

.done 是一个成功的回调验证，类似于 readyState=4 和 request.status=200；对此，可参考第 3 章中的相关内容，其间，曾利用 JavaScript 生成异步调用

对应的输出结果如图 4.2 所示。

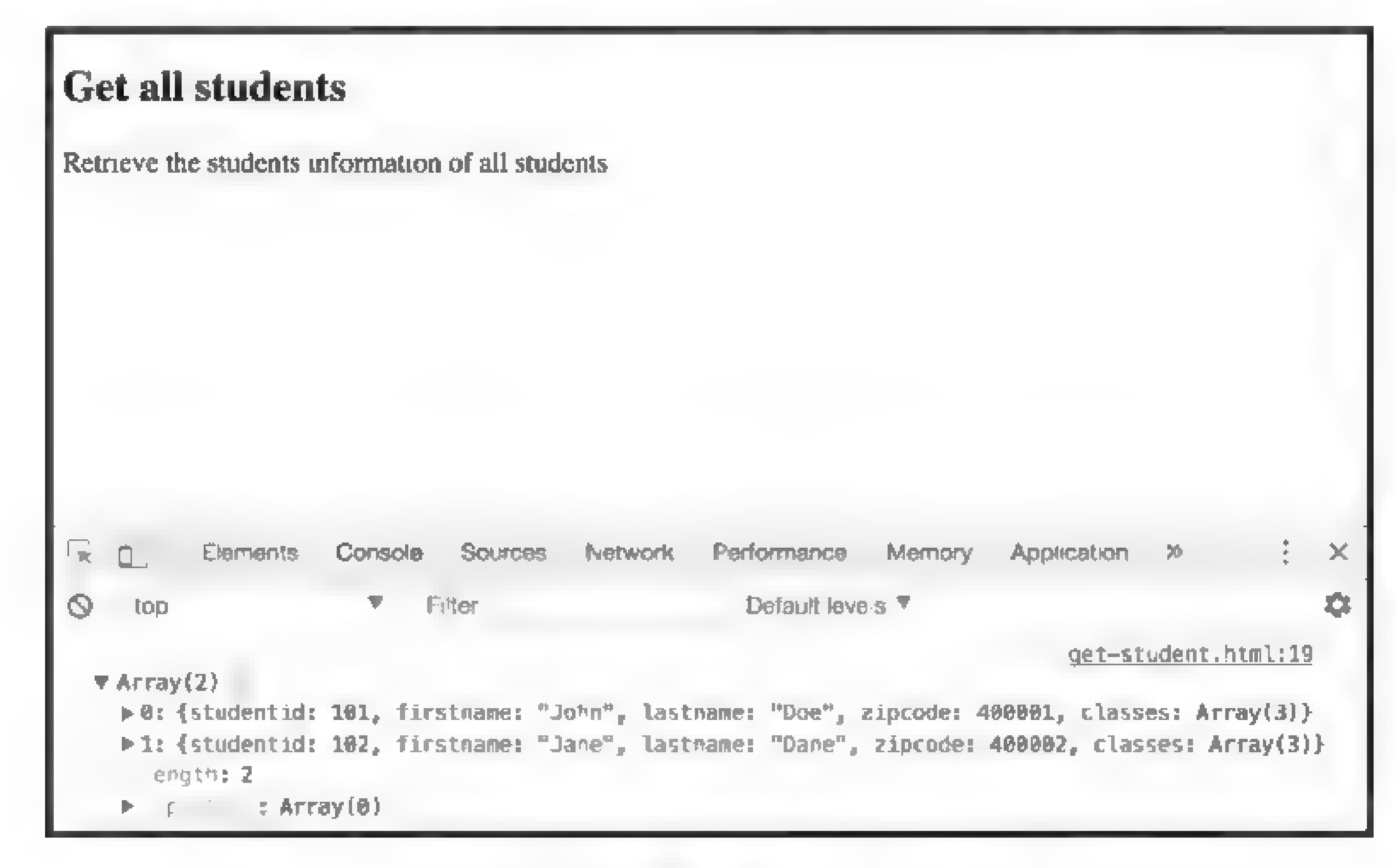



图　4.2

在 Console 窗口中，可以看到源自服务器的 JSON 提要响应结果。该 JSON 提要包含了大量信息——对应数据来自所有的学生。接下来将根据邮政编码获取学生记录。针对当前示例，我们将使用 zipcode 参数，并通过 HTTP GET 方法将一个值异步传递给服务器。这一API 调用将为数据供应商提供如下服务：这些供应商希望搜索特定区域内的实习生。

```
$.ajax({
  "url": "http://localhost:3300",
  "type": "GET",
  "data": {"zipcode": "400001"},
  "dataTYPE":"JSON"
})
.done((data)=>{
  console.log(data);
})
```

代码首先导入了 jQuery 库，然后绑定一个回调函数，以准备在加载文档时所触发的事件。需要注意的是，此处采用了 data 属性发送邮政编码的键-值对，如图 4.3 所示。

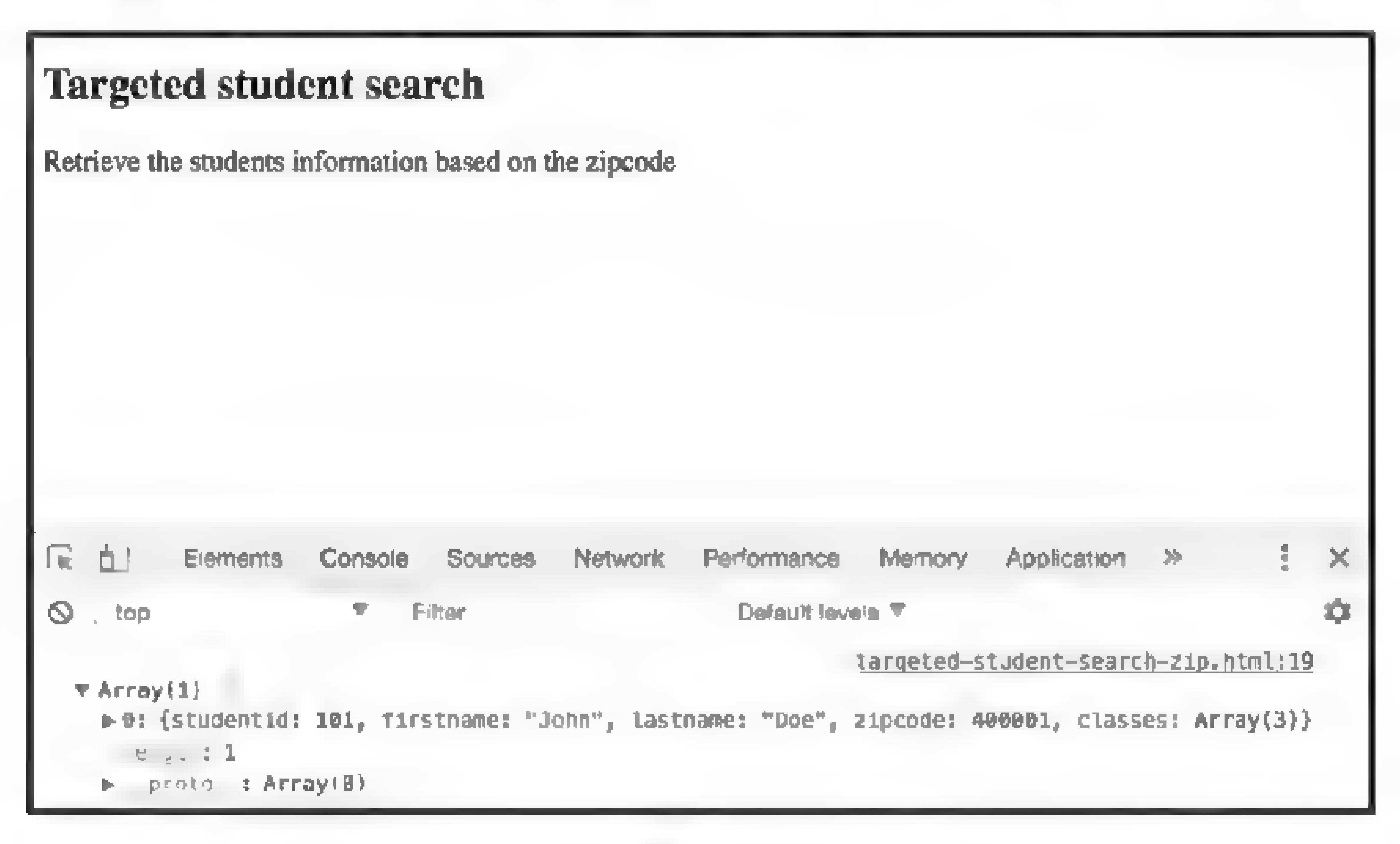



图　4.3

当调用被触发后，响应结果将输出至 Console 窗口中。其中，邮政编码 400001 匹配了一个用户，学生信息则通过 JSON 提要传回。相应地，有针对性地搜索可帮助我们缩小搜索范围，进而提供所搜索的数据。下一个目标搜索将采用学生注册的课程检索数据。

对此，可将邮政编码搜索模式切换为学生注册的课程，如下所示。

```
http://localhost:3300/?class Economics
```

在该示例中，URL 将返回注册了课程 Economics 的全部学生信息。在服务器端，需要对包含 case'/'的 URL 进行修改，如下所示。

```
case '/':
  let student = studentsData.filter((student) =>{
    if(queryObject.zipcode){
       return (student.zipcode == queryObject.zipcode);
    }
    else if(~((student.classes).indexOf(queryObject.class))){
      return student;
    }
  })
  if(student.length==0){
    student = studentsData;
  }
  res.write(JSON.stringify(student));
  res.end();
break;
```

在客户端，则需要在浏览器中运行下列 HTML 代码。

```
<!DOCTYPE html>
<html>
<head>
  <title>Targeted student search</title>
  <!-- Go to https://code.jquery.com/ for more details -->
  <script src="https://code.jquery.com/jquery-3.2.1.min.js"
    integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
    crossorigin="anonymous"></script>
  <script>
  $(document).ready(() => {
    $.ajax({
      "url": "http://localhost:3300",
      "type": "GET",
      "data": {"class": "Economics"},
      "dataType":"JSON"
    })
    .done((data) =>{
      console.log(data);
    })
  })
  </script>
</head>
<body>
```

```
  <h2>Targeted student search</h2>
  <p>Retrieve the students information based on classes thy are enrolled
in.</p>
</body>
</html>
```

该示例基本等同于基于邮政编码的目标搜索，此处利用课程信息替换了邮政编码信息。相应地，这将检索注册了课程 Economics 的全部学生，对应结果如图 4.4 所示。

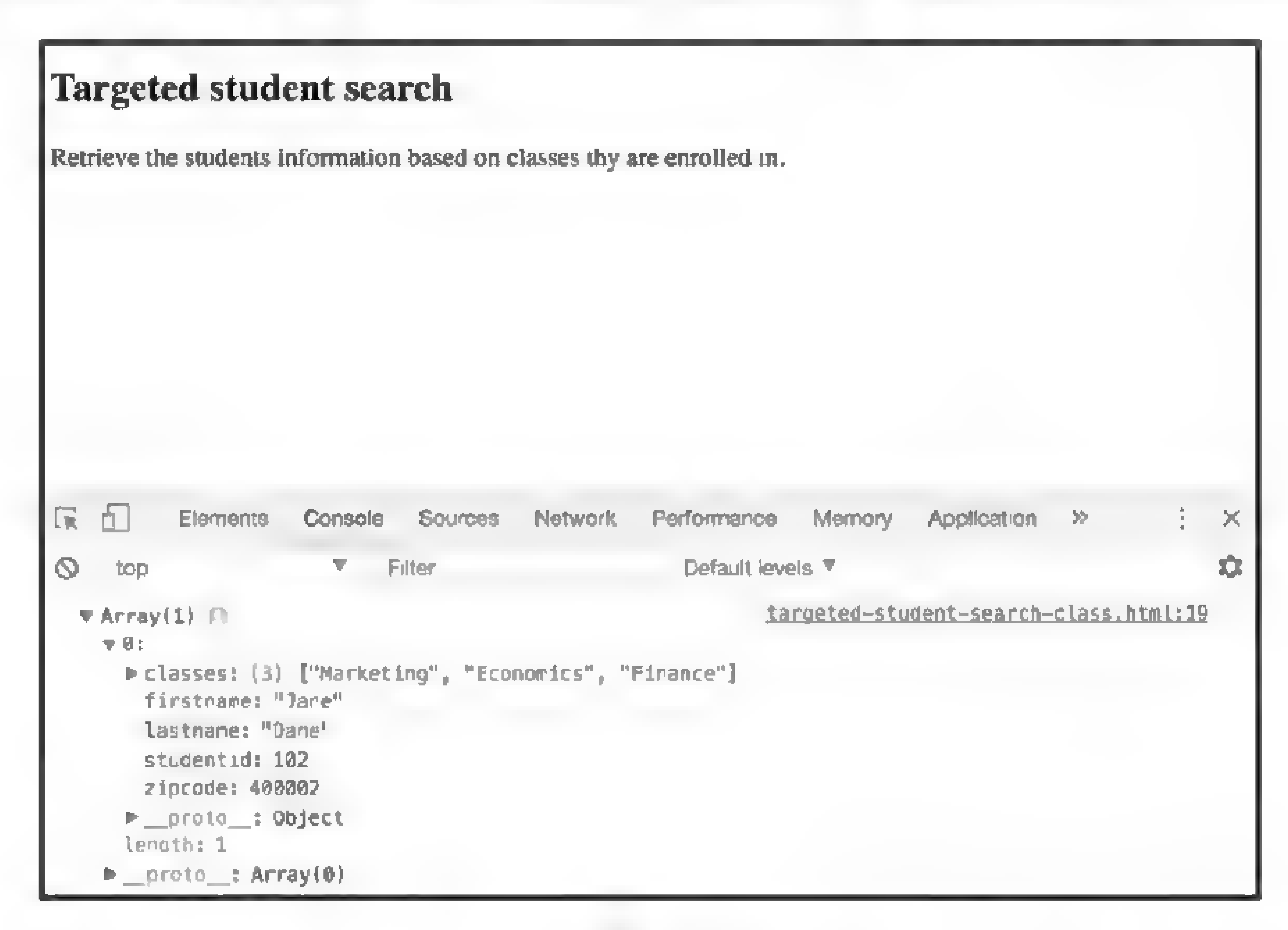



图　4.4

前述内容讲述了通过 HTTP GET 方法生成异步调用的多个示例，接下来将把对应数据推送至服务器，以便利用 API 添加学生信息。API 中的最后一个调用将由用于添加学生的 HTTP POST 方法提供支持。

POST 请求方法常用于发送较大的数据。与 GET 方法不同，数据将通过 HTTP 消息体进行传输。我们可采用诸如 Fiddler，或通过浏览器中的开发工具跟踪途经消息体的数据。通过 POST 方法传递的消息无法添加书签或被缓存，这一点与 GET 方法不同。POST 方法通常用于在使用表单时发送数据。

当处理服务器端的 POST 请求时，需要在节点服务器中进行适当的调整，并在 switch 语句中添加一条 case '/addUser':语句，如下所示。

```
case '/addUser':
  let jsonString = '';
  req.on('data', (chunk) => {
    jsonString += chunk;
  });
  req.on('end', () => {
    let parseJSON = JSON.parse(jsonString);
    studentsData.push(jsonString);
    res.end(JSON.stringify(jsonString));
  });
break;
```

下面简要地介绍一下服务器端的数据处理。

- 默认状态下，不会在请求参数中直接接收体信息。相反，需要在请求对象 data 监听器上监听所请求的数据，这意味着将在事件 data 监听器上收集请求数据。一旦数据从请求处收集完毕，将会自动触发请求 end 事件。
- 事件 end 监听器的回调执行简单的操作，即将数据推送至 student 数组中，并通过将整个 student 数据发送回客户端进行响应。这里应确保将 studentsData 的标识符关键字从 const 更改为显示导入时使用的关键字。

所以，回到客户端，POST 请求所需的 URL 如下所示。

```
http://localhost:3300/addUser
```

作为 HTTP POST 方法，所传递的数据处于不可见状态。下面将通过脚本访问此类调用。其中，第一个脚本将访问提供所有学生信息的 API 调用。

我们将使用 addUser 调用动态地添加一名学生。据此，开发团队可利用外部资源将学生信息添加至数据库中。例如，假设作为一个学生信息聚合器，我们向多家数据供应商出售整合后的学生信息。当聚合此类信息时，可能需要采用 Spider（脚本将访问网站并获取数据）或外部资源（数据于其中呈现为非结构化状态）实现这一任务。因此，我们将结构化数据，并使用这一 addUser API 调用将结构化的学生数据信息摄取到数据存储中。同时，可以将此方法公开给受信任的数据供应商，以使我们的数据存储成为一个单点数据位置。这对两家公司来说是一个双赢的结果：我们可获得更为丰富的学生信息；数据供应商则将学生信息存储在远程位置处。

下列代码展示了 addUser POST 调用的生成方式。

```
//add-user.html
<!DOCTYPE html>
<html>
<head>
```

```
  <title>Adding a student</title>
  <!-- Go to https://code.jquery.com/ for more details -->
  <script src="https://code.jquery.com/jquery-3.2.1.min.js"
  integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
  crossorigin="anonymous"></script>
  <script>
  $(document).ready(() => {
    const first name = "kent";
    const last name = "clark";
    const addresses = ["5400 W Parmer Ln", "1919 Elridge Pkwy"];
    const zipcodes = ["78757", "77887"];
    const classes = ["International Business", "Economics Statistics"];
    $.ajax({
      "url": "http://localhost:3300/addUser",
      "type": "POST",
      "data": {
        "first name": first name,
        "last name": last name,
        "addresses": addresses,
        "zip codes": zipcodes,
        "classes": classes
      },
      "content-type": "application/json; charset.utf-8",
      "dataType": "JSON"
    }).done(function(data) {
      console.log(data);
    });
  });
  </script>
</head>
<body>
  <h2>Adding a student</h2>
  <p>Storing the student information</p>
</body>
</html>
```

上述调用执行了多项任务。首先，我们声明了一些变量加载本地数据。具体来说，局部变量将加载学生姓名的字符串值，其他一些变量则加载课程数组、邮政编码和地址。在 AJAX 调用中，第一个值得注意的变化是类型属性。由于将推送大量的用户数据，一般会使用 HTTP POST 方法。data 属性将使用为名称、姓氏、地址、邮政编码和类声明的局部变量。在 API 中，当用户成功地添加到数据库时，我们在响应中发送一个学生列表，该列表将被输出到 Console 窗口中。

当验证新的'student'是否已被添加至数据库时，可检测最后一个数组元素的响应结果，或者运行 getStudents API 调用以查看完整的用户列表。

在学生提要中的最后一名学生是'Kent Clark'——测试代码以确保一切正常工作是十分重要的。在处理动态数据时，维护数据完整性同样非常重要。每当对用户或其依赖项执行 CRUD 操作时，必须通过查看检索到的数据，以及执行数据验证检测，进而验证该数据存储上的数据完整性。

## 4.3　跨域 AJAX 调用存在的问题

截止到目前，所有的异步调用均位于阶段服务器上。其中，节点服务器负责处理基于 CORS 的各项功能。在某些情况下，我们希望从不同的域加载数据，例如，从其他 API 获取数据，这些 API 可能不支持 CORS。服务器端程序设计可用来处理这类调用；相应地，可以使用 cURL 并针对不同的域进行 HTTP 调用，从而获取此类数据。由于需要对服务器生成调用，而服务器又调用另一个域来获取数据，并返回客户端程序，因而这增加了我们对服务器端应用程序的依赖。或许这并不是一类严重的问题，但此时正在向 Web 架构中添加一个附加层。为了避免生成服务器端调用，可尝试对不同的域执行异步调用。例如，可采用学生的 JSON API 获取数据。

由于在节点服务器上处理了 CORS，因而需要在节点服务器项目中对 app.js 做如下修改。

```
res.writeHead(200, {
  "Content-Type": "application/json",
  //"Access-Control-Allow-Origin": "*"
});
```

在上述代码片段中，我们注释掉了 Access-Control-Allow-origin 选项，用于模拟未经处理的跨源资源请求。随后运行 HTML 代码，如下所示。

```
<!--get-student-cor.html-->
<!DOCTYPE html>
<html>
<head>
  <title>Get all students</title>
  <!-- Go to https://code.jquery.com/ for more details -->
  <script src="https://code.jquery.com/jquery-3.2.1.min.js"
    integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
    crossorigin="anonymous"></script>
  <script>
  $(document).ready(() => {
```

```
    $.ajax({
      "url": "http://localhost:3300",
      "type": "GET",
      "data": {},
      "dataTYPE":"JSON"
    })
    .done((data)=>{
      console.log(data);
    })
  })
  </script>
</head>
<body>
  <h2>Asynchrnous call for student api</h2>
  <p>Cross origin request neither handled at server not at client
    side</p>
</body>
</html>
```

当跨域生成上述异步调用时，对应的输出结果如图 4.5 所示。

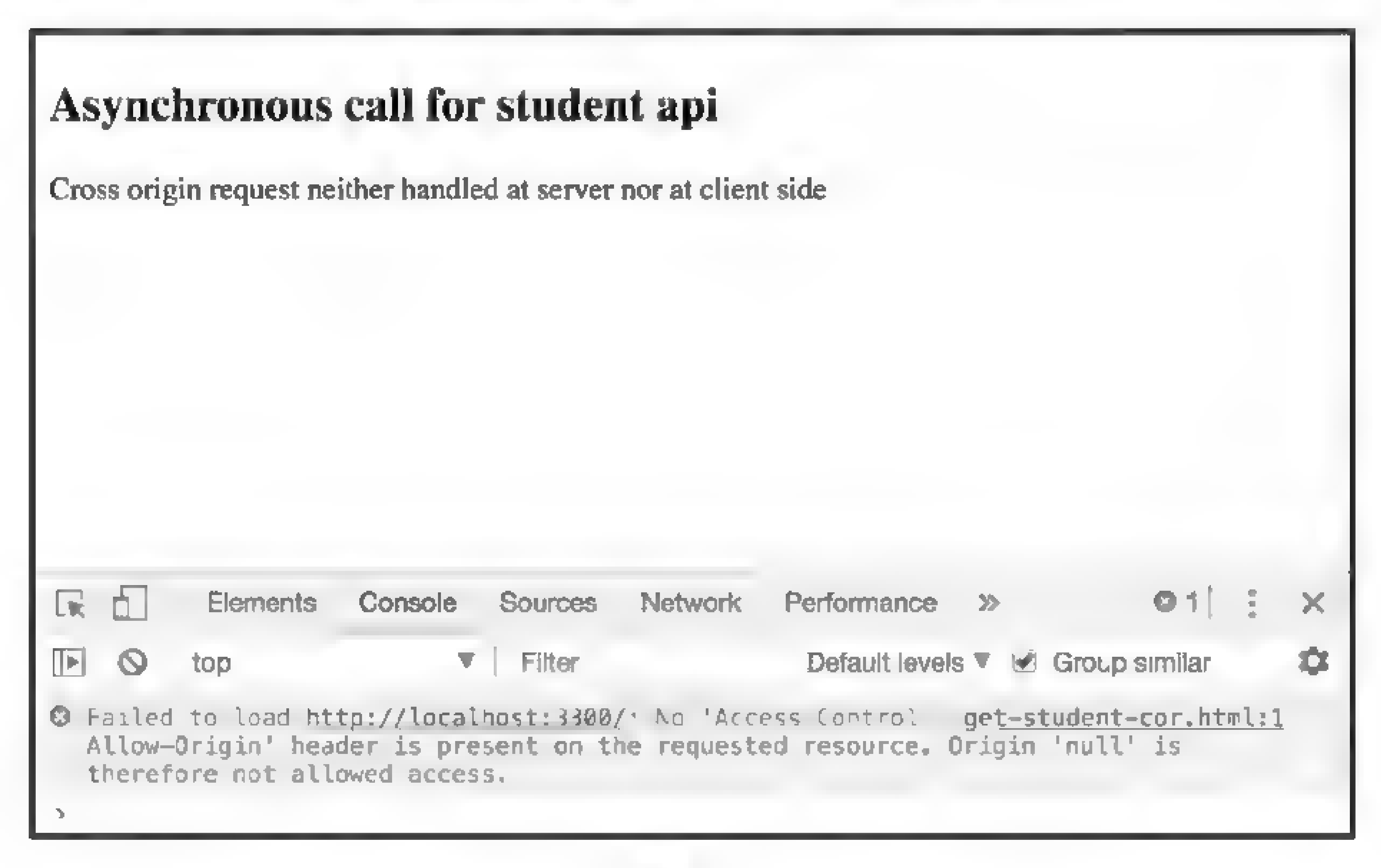



图　4.5

在当前异步调用中，Console 窗口中显示了一条错误消息，表示 XMLHTTPRequest 对象不能加载我们提供的 URL，因为它不是来自浏览器的当前域，这导致我们违反了源

协议。同域策略是 Web 浏览器遵循的安全措施，目的是防止一个域访问另一个域上的信息。Web 应用程序使用 cookie 存储关于用户会话的基本信息，以便在用户下一次请求相同的 Web 页面，或在相同域中请求不同的 Web 页面时提供直观的用户体验。为了防止外部网站窃取这些信息，Web 浏览器遵循同源策略。

同域策略在输入的请求中查询 3 项事务，即主机、端口和协议。为了减缓此类问题，下面讨论一种称为 JSONP 的 CORS 处理方法。

# 4.4　JSONP 简介

当处理同源策略这一类问题时，可采用 JSONP（基于填充的 JSON）方案。同源策略下的一个例外是<script>标签，因此脚本可以跨域传递。JSONP 据此将数据作为脚本跨域传递，方法是通过填充方式使 JSON 对象看起来像是一个脚本。在 JavaScript 中，当调用一个包含参数的函数时，我们可调用该函数并添加一个参数；而采用 JSONP 时，可作为参数向函数传递一个 JSON 提要，因而将对象填充至函数回调中。相应地，填充了 JSON 提要的函数须用于客户端，进而检索 JSON 提要。图 4.6 显示了一个 JSONP 示例。

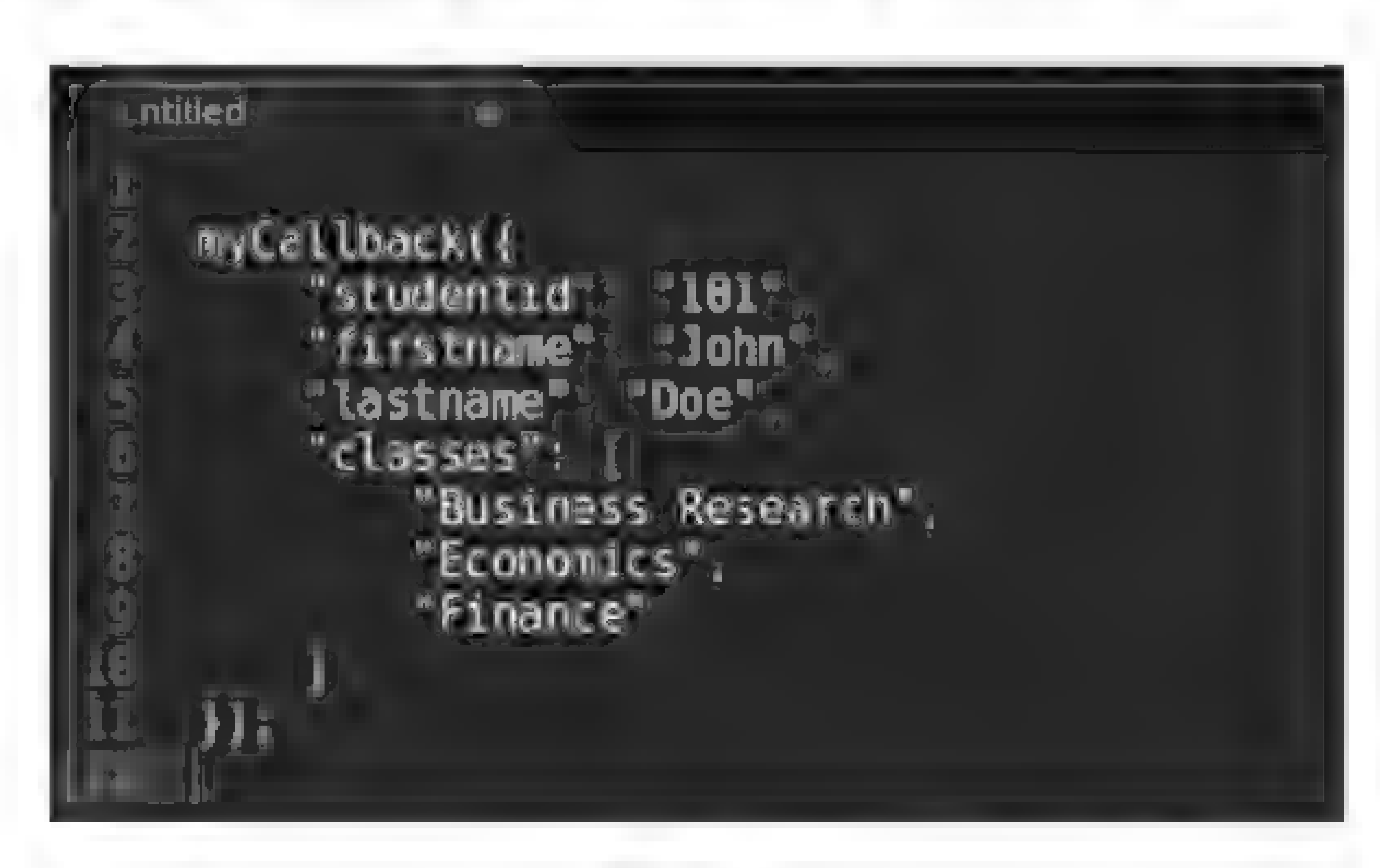



图　4.6

在该示例中，我们向 myCallback()函数中填充了学生对象，并复用 myCallback 以检索学生对象。在理解了 JSONP 的工作方式后，接下来将该技术应用于应用程序中。当与 JSONP 方案协同工作时，还需要得到服务器端代码的支持。下面讨论服务器端的一些变化内容。

## 4.4.1　服务器端实现

在处理 JSONP 请求时，需要在服务器端实现下列过程。

（1）在 switch 代码块中添加一个 case，以表示一个 JSONP 请求，查看下列代码。

```
case '/.jsonp':
break;
```

（2）一旦“路由”case 定义完毕后，首先需要实现一个查询解析逻辑。在当前示例中，已经通过 querystring.parse()方法解析了查询对象，这将从 URL 获得作为查询参数传递的回调详细信息。随后需要生成一个 JSONP 响应：首先需要将 JSON 响应字符串化，然后将其填充到从客户端接收到的 JSONP 回调中，如下所示。

```
case '/.jsonp':
//validate query parameter
const jsonpCallback = queryObject.jsonp;
if(!jsonpCallback){
  return res.end(studentsData);
};
let start = jsonpCallback + '(', end = ')';
let stringifiedStudentData = JSON.stringify
  (studentsData, undefined, 2);
res.end(start + stringifiedStudentData + end);
break;
```

上述代码提供了查询 URL，即 http://localhost:3300/.jsonp?jsonp=getStudentData，当在浏览器中执行单击操作时，将提供一个 JSONP 响应。接下来将考查 JSONP 在浏览器端的实现。

### 4.4.2　在客户端（浏览器）实现 JSONP

下面利用新创建的脚本（获取 JSON 提要）替换之前脚本中的 URL 属性。其间，诸如 URL 和 dataType 这一类属性将被修改，同时加入了 contentType 和 jsonpCallback 这一类新属性。前述内容已经讨论了 URL 属性的变化，接下来查看一下其他属性，如下所示。

```
<!DOCTYPE html>
<html>
<head>
  <title>Asynchronous Call to Reddit</title>
  <script src="https://code.jquery.com/jquery-3.2.1.min.js"
    integrity="sha256-hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
    crossorigin="anonymous"></script>
  <script>
  $(document).ready(() => {
    $.ajax({
```

```
      "url": "http://localhost:3300/.jsonp?jsonp=getStudentData",
      "type": "GET",
      "data": {},
      "dataType": "JSONP",
      "jsonpCallback": "getStudentData"
    })
    .done((data) => {
      console.log(data);
    })
  })
  </script>
</head>
<body>
  <h2>Asynchronous call for Students api</h2>
  <p>Cross request is not handled at server but at client</p>
</body>
</html>
```

之前，由于输入的提要为 JSON 类型，因而 dataType 属性被设置为 JSON；当前，JSON 提要被填充至一个回调中，因而应做适当调整以体现浏览器期望一个回调，而非 JSON 自身。其中，contentType 和 jsonpCallback 则是所加入的新属性；属性 contentType 用于指定发送至 Web 服务器的内容类型；jsonpCallback 将接收一个回调函数名，JSON 提要将填充于其中。

当脚本被触发后，源自 getStudentData 回调中的数据将被检索，并传递到 success 属性中，同时将 JSON 对象记录至控制台窗口中，对应的输出结果如图 4.7 所示。

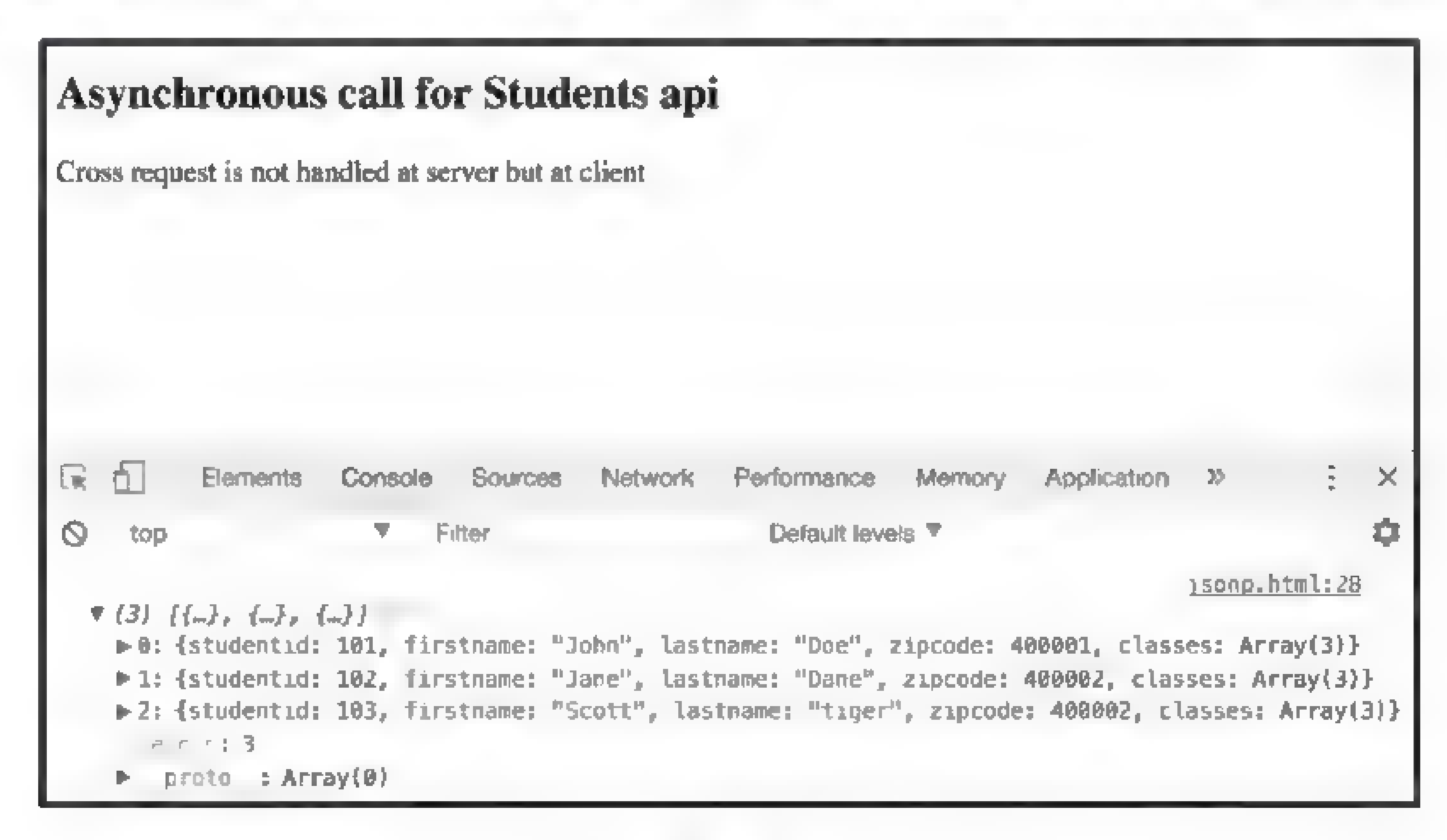



图　4.7

注意，JSONP 调用是一个脚本调用，而非 XHR 请求，因此 JSONP 调用将出现于 JS 或<scripts>标签中，而不是 Console 窗口的 XHR 选项卡中 此外，JSON 通常是一个 GET 方法请求。

## 4.5 本 章 小 结

HTTP GET 和 POST 方法是两个较为流行的 HTTP 方法，并在客户端和服务器间传输数据。本章深入讨论了基于异步请求的 GET 和 POST 请求方法的数据传输方式。随后，本章介绍了跨源资源问题及其解决方案。另外，本章还探讨了 JSONP 在客户端和服务器端的实现。

关于跨域异步请求，我们使用了<script>标签执行 JSONP 异步脚本调用，以获取来自不同域中的数据。第 5 章将利用各种客户端工具对 JSON 进行调试。

# 第 5 章　JSON 调试

JSON 在过去几年中飞速发展，因此有大量免费可用的资源可以帮助我们理解正在使用的 JSON 对象。如前所述，JSON 可用于多种功能；另外，一些令人忽视的内容往往会引发 JSON 中断，例如双引号、JSON 对象中最后一项上的逗号，或者 Web 服务器发送的错误的内容类型。本章主要涉及以下主题。

- 快速熟悉开发工具。
- 验证 JSON。
- 利用一些网站格式化 JSON。

## 5.1　使用开发工具

几乎所有的浏览器均配置了功能强大的调试工具，以帮助我们理解生成的请求，以及所返回的响应结果，例如 Mozilla Firefox、Google Chrome、Safari 和 IE。相应地，JSON 既可以是请求中的部分内容，也可以是响应结果中的部分内容。Google Chrome、Safari 和较新版本的 IE 均内建了开发工具。对于 Mozilla Firefox 来说，Firebug 则是十分常用的 Web 开发工具箱。Firebug 是一个外部插件，并可安装在浏览器中。这也是最早的 Web 开发工具箱，旨在向 Web 开发人员提供各种帮助。最近，Mozilla Firefox（即 Firefox Developer Edition）发布了最新的前端开发工具，其中包括开发工具箱和开发环境。

Firefox 可对 HTML DOM 树进行访问，并可动态地查看 HTML 元素在页面上排列方式。另外，浏览器还提供了一组开发工具，并可模拟客户端上的 Web 开发过程。下列内容列出了其中的一些特性。

- 在线查看器：可通过在线方式方便地查看 HTML、CSS 和 JS 代码。
- JavaScript 调试器：配置了 Redux 和 React 等最新工具。
- Network 选项卡：可跟踪全部资源，例如图像、JavaScript 文件、CSS 文件、Flash 媒体以及客户端生成的异步调用。除此之外，还可对延迟进行监视。
- Console 窗口：这是另一个较为常用的工具。顾名思义，该窗口提供了运行期内的 JavaScript 控制台信息，并可以动态方式对脚本进行测试。

当加载 Firefox（Firefox Developer Edition）、Google Chrome 和 Safari 的开发工具时，可在 Web 页面上右击，随后从选项列表中选择 Inspect Element 选项。

当与 Safari 协同工作时，须启用开发工具，具体如下。

- ❑ 单击 Safari 菜单选项。
- ❑ 选择 Preferences 并单击 Advanced 选项卡。
- ❑ 在菜单栏中选择 Show develop menu，并可对开发工具进行查看。

对于 IE，按键盘上的 F12 键并打开开发工具窗口。在第 3 章中，我们曾对 Web 服务器生成了第一个异步调用，并利用 jQuery 请求 JSON 数据。接下来，我们将在 Firefox Developer Edition 的浏览器中调试此类静态 HTML 文件。

开始时，需要启动 Node 服务器，以便客户端脚本中的 jQuery Ajax 调用可接收数据。对此，可使用 node-test-app 目录中的命令（参见第 3 章），如下所示。

```
npm start
```

在浏览器中打开 jquery-ajax.html 文件，并利用开发工具调试数据。当采用 Firefox Web 浏览器时，对应结果如图 5.1 所示。




图 5.1

在加载页面上，右击并选择 Inspect Element，默认状态下将加载 html 选项卡，用户将看到 HTML DOM。在当前示例中，我们向 Get Feed 按钮上绑定了一个单击事件处理程序。在单击该按钮后，可查看 Console 输出结果。对此，可单击 Console 选项卡，如图 5.2 所示。

一旦检索到响应结果，JSON 提要即记录至 Console 窗口中。本书主要采用 console.log 方法向 Console 窗口中输出数据，这也是开发工具中十分有用的一个特性，这对于理解经




图　5.2

解析、显示后的 JSON 提要来说十分重要。除此之外，Console 窗口还提供了一种简单方式以对 JSON 提要进行分析。接下来将访问 Firefox 中的 Network 选项卡，进而了解客户端所期望的、客户端和服务器间的内容类型的通信方式，如图 5.3 所示。




图　5.3

在 Network 窗口中，首先单击异步调用的 URL，即 http://localhost:3300/。单击该链接后，在 Headers 部分中可以看到 Accept 头，其期望内容为 application/json。注意，左侧栏

中显示了两个请求，分别是脚本请求和 XHR 或 XMLHttpRequest，并通知这是一个异步请求。Network 窗口右侧的 Response 选项卡中将显示针对该请求的 JSON 提要。

## 5.2　验证 JSON

类似于对资源进行调试，存在大量的 Web 工具可帮助我们验证所构建的 JSON。其中，JSONLint 是一种较为常用的 Web 工具，并可对 JSON 提要进行验证。

ⓘ 当与 JSONLint 协同工作时，可访问 https://jsonlint.com/以了解更多信息。

JSONLint 包含了一个较为直观的界面，以使用户可粘贴需要验证的 JSON，随后根据 JSON 提要返回一条成功消息，或一条错误消息。下面首先验证一个包含某种问题的 JSON，进而查看所返回的错误消息；随后将对此进行修复，并再次查看成功消息。

对此，可复制前述示例中的 students 提要，并在第二个元素的结尾处添加一个逗号，如图 5.4 所示。

```
[{
    "studentid": 101,
    "firstname": "John",
    "lastname": "Doe",
    "classes": ["Business Research", "Economics", "Finance"]
}, {
    "studentid": 102,
    "firstname": "Jane",
    "lastname": "Dane",
    "classes": ["Marketing", "Economics", "Finance"]
}, ]

Validate JSON   Clear          Support JSONLint for $2/Month

Results

Error: Parse error on line 11
...ics", "Finance"]}, ]
                      ^
Expecting 'STRING', 'NUMBER', 'NULL', 'TRUE', 'FALSE', '{', '[', got ']'
```

图　5.4

注意，此处在 JSON 对象的最后一项处添加了一个逗号。JSONLint 提供了描述性错误信息，这也是 JSONLint 的优点之一。当前，我们遇到了一个 Parse error，为了简单起见，该消息还提供了错误所在的行号。这里，解析器期望一个字符串、一个数字、一个 null 或一个布尔值；由于我们未提供任何内容，因而将生成一条错误信息。对此，可向 JSON 对象添加一个新项来证明逗号的正确性，或者去掉逗号，因为前面没有任何项，如图 5.5 所示。




图　5.5

在移除了逗号并再次验证时，我们将得到一条成功消息。易用性和描述性消息使得 JSONLint 成为 JSON 验证的首选工具之一。

## 5.3　格式化 JSON

JSONLint 不仅是一个在线 JSON 验证器，还可以帮助我们格式化 JSON，以使其看上去更加美观。JSON 通常较大，在线编辑器需要提供一种树形结构以遍历 JSON 对象。相

应地，JSON Editor Online 即是一种深受喜爱的在线编辑器，并可格式化较大的 JSON 对象，它提供了一种便于浏览的树形结构，如图 5.6 所示。

图　5.6

当与 JSON Editor Online 协同工作时，可访问 https://jsoneditoronline.org/以了解更多信息。

首先可将 JSON 示例代码粘贴到左边的窗口中，然后单击中间的右箭头按钮来生成树形结构。一旦对树形结构进行了更改，即可单击左箭头按钮来格式化数据，使其可以在其他地方使用。

## 5.4　本 章 小 结

调试、验证和格式化是开发人员不可忽视的 3 项任务。本章介绍了浏览器开发工具及其使用方式，并可用于调试任务。此外，我们还分别讨论了 JSONLint 和 JSON Editor Online，以用于验证和格式化操作。

本章是 JavaScript 和 JSON 基础知识的最后一部分内容，目的是让读者深入了解如何以 JSON 数据格式存储和传输数据。其中涉及在同一域内通过 HTTP 异步请求传输 JSON，以及跨域传输 HTTP 异步请求。除此之外，我们还研究了 JSON 数据格式应用方式的其他实现。这也为理解 JSON 以开发交互式和响应性 Web 应用程序打下了坚实的基础。第 6 章将具体运用我们学习的前端脚本知识开发一个 Carousel 应用程序。

# 第 6 章　构建 Carousel 应用程序

前述章节花费了大量的篇幅介绍 JavaScript 和 JSON。本章将尝试构建基于 JSON 的端到端项目，其间将涉及 JavaScript、JSON 等多种概念、服务器端程序设计、AJAX 和 JSONP，并对此进行适当的整合。本章主要涉及以下主题。

- 配置应用程序。
- Bootstrap 简介。
- 维护 JSON 存储。
- 利用 jQuery Cycle 实现 Carousel 功能。

本章将设计一个轮播提示板应用程序，以展示当月的优等生。该应用程序提供了轮播功能，其中包括导航按钮、内容的自动播放、在既定点显示独立项，并跟踪内容的首、尾项。

## 6.1　配置 Carousel 应用程序

对此，首先需要配置一个文件夹，并加载该应用程序的相关文件。具体来说，该应用程序需要使用一个 HTML 文件加载 Carousel，以及诸如 jQuery 和 jQuery Cycle 等库（因而需要导入这些库）。除此之外，还需要使用加载数据的 JSON 文件。当下载 jQuery 文件时，可访问 https://jquery.com。如前所述，jQuery 是一种较常使用的 JavaScript 库，其社区也处于不断壮大中。同时，本章将使用 jQuery Cycle 库，以进一步丰富 Carousel 应用程序的各项功能。jQuery Cycle 是最流行的轻量级循环库之一，具有许多特性，如响应组件，以及针对按钮和节流速度的可配置交互行为。读者可访问 http://malsup.github.io/jquery.cycle.all.js 下载 jQuery Cycle，并将对应文件重命名为我们使用的 jquery.cycle.js。

接下来将生成一个名为 chapter 6 的目录，并将全部下载文件存储于其中，对应结构如图 6.1 所示。

至此，文档根目录中已经配置了相关库。接下来将处理这些基本的 HTML 文件，并将此类文件导入 Web 页面中，如下所示。

```
//index.html
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript" src="jquery-3.2.1.min.js"></script>
```

```
  <script type="text/javascript" src="jquery.cycle.js"></script>
  <script type="text/javascript">
  $(document).ready(() => {
    console.log("Ready!");
  })
  </script>
</head>
<body>
</body>
</html>
```

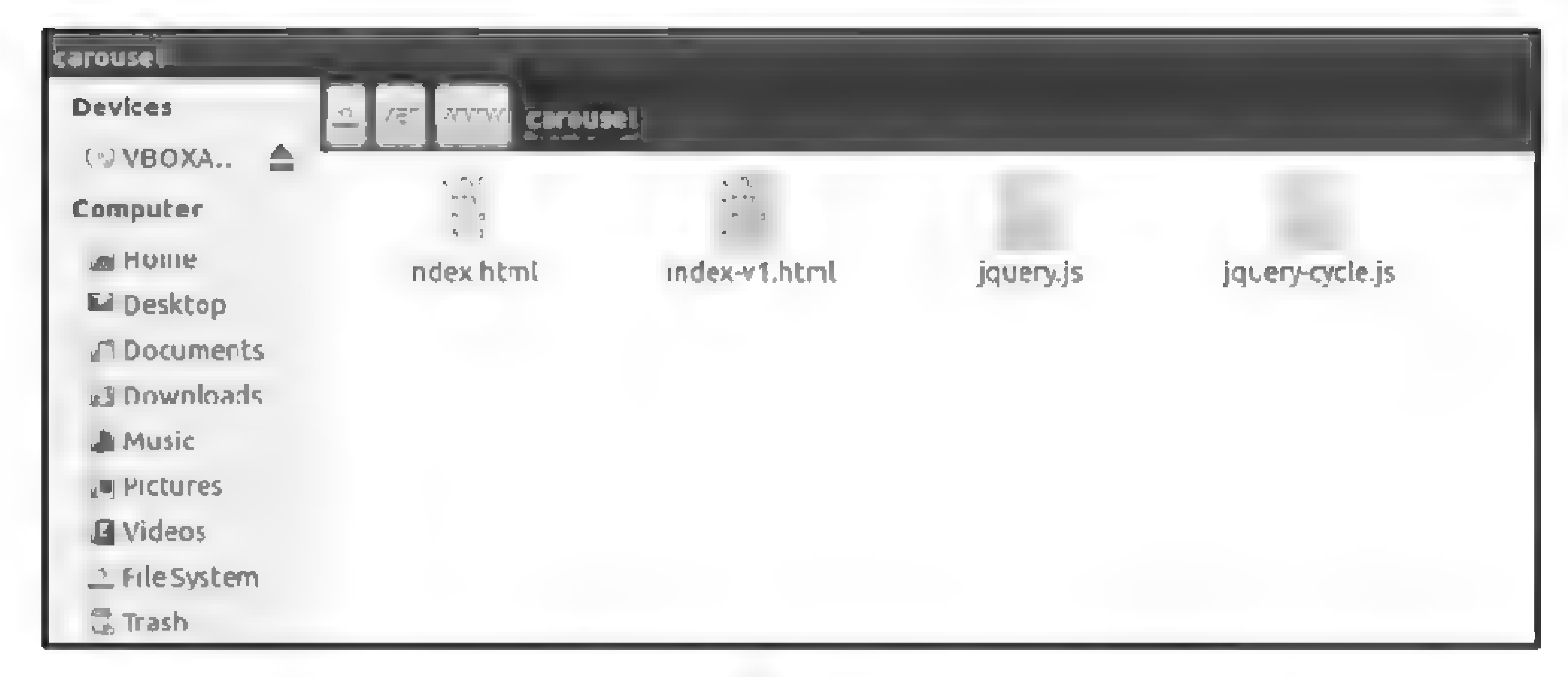



图 6.1

这表示为最初的索引 Web 页面，并将 JavaScript 文件加载至 Web 页面中。当通过 Web 浏览器触发该文件时，两个 JavaScript 库将被加载并准备就绪，相关消息将输出至 Console 窗口中。接下来将要处理数据文件，这与之前讨论的学生的 JSON 提要基本类似，随后将其加载至一个旋转应用程序中。

## 6.2 生成 Carousel 应用程序的 JSON 文件

假设我们是一家教育机构，按照惯例，每个月都要表彰一些努力用功的学生。每个月内，我们将对每门课程选取一些排名靠前的学生，并在轮播提示板上显示他们的姓名，进而激励其他学生，这也是一些教育机构经常采用的一种做法。图 6.2 显示了相应的 JSON 提要内容。

轮播提示板应用程序需要使用基本的学生信息，例如名字、姓氏、当前教育水平，以及表现优异的课程，如下所示。

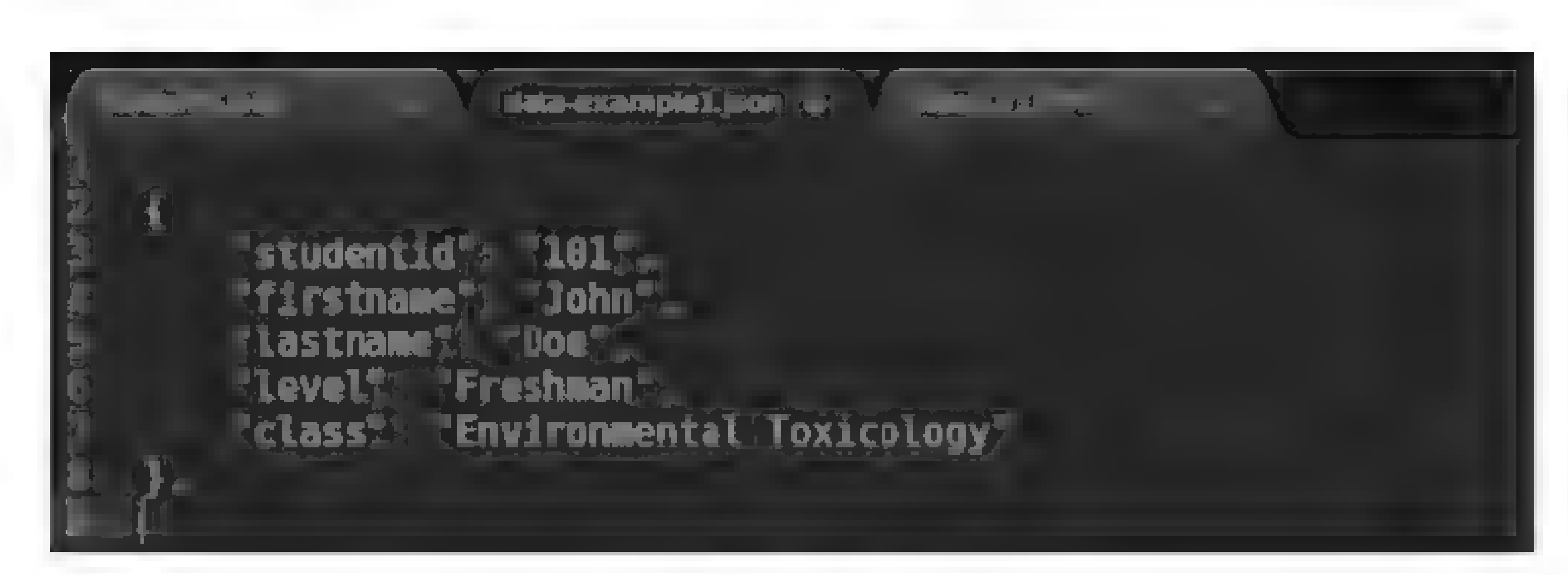



图　6.2

```
<script type="text/javascript">
$(document).ready(()=>{
  $.getJSON('http://localhost:3000', (data)=>{
    console.log("data on ", data);
  })
})
</script>
```

上述代码片段使用了 jQuery 的 getJSON()函数将 JSON 提要置入文档中。当 index.html 文件被载入浏览器中，学生的 JSON 对象数组也将被载入 Console 窗口中。接下来，将析取 JSON 对象中的数据，并将其嵌入 DOM 中。对此，可采用 jQuery 的 each()函数遍历学生和 JSON 提要，并将数据加载至当前页面中。

jQuery 的 each()函数类似于服务器端的 foreach()迭代循环和原生 JavaScript 的 for in()迭代循环。其中，迭代器 each()将数据作为第一个参数，并将数据中的各项作为一个键-值对传递至回调中。这里，回调表示为一个执行于键-值对上的脚本集合。在该回调中，我们将构建一个 HTML 文件，而该文件将追加到 DOM 上的 div 元素上。针对学生 JSON 对象中的所有元素，我们将使用这一回调并以迭代方式构建 HTML 文件，如下所示。

```
<script type="text/javascript">
$(document).ready(() => {
  let htmlContent = ``;
  $.getJSON('http://localhost:3300', (data) => {
    $.each(data, (key, value) => {
      $.each(value, (index, student) => {
        htmlContent += `<div class="student">`;
        htmlContent += `<h3>${student.level} of the Month</h3>`;
        htmlContent += `<h4>${student.firstname}
        ${student.lastname}</h4>`;
        htmlContent += `<p>${student.class}</p>`;
        htmlContent += `</div>`;
      });
```

```
    });
    $('#students').html(htmlContent)
  })
})
</script>
```

在 index.html 文件中，我们采用了 jQuery 中的 each()函数遍历学生的 JSON 提要并构建 HTML 文件，以显示学生信息，例如名字、姓氏、入学年份和所选课程。这里将构建一个动态 HTML 并将其赋予当前 html 变量中。相应地，html 变量中的数据将于稍后添加至包含学生 ID 的 div 元素中，如下所示。

```
<body>
  <div id="students"></div>
</body>
```

图 6.3 显示了 index.html 体的输出结果。

**Freshman of the Month**

**John Doe**

Environmental Toxicology

**Senior of the Month**

**Jane Dane**

Economics

**expert of the Month**

**Scott tiger**

Physics

图　6.3

当脚本被载入 Web 浏览器后，脚本将检测当前文档是否处于就绪状态。若是，将向服务器生成一个 AJAX 调用，进而检索 JSON 数据。在 JSON 检索完毕后，学生 JSON 对象数组中的每个对象将被传递至回调中，并生成一个包含学生课程的 HTML div 元素。这一过程持续进行，直至回调运行于最后一个元素。此时，HTML 文件将被追加至 HTML

中的 div 元素上（包含一名学生的 ID）。

关于 HTML 追加于 htmlContent 变量上，这里可稍作更改，如下所示。

```
$.each(data, (index, student)=>{
  htmlContent+=`<div class="student">
    <h3>${student.level} of the Month</h3>
    <h4>${student.firstname} ${student.lastname}</h4>
    <p>${student.class}</p>
  </div>`;
});
```

与之前的脚本相比，此处并未针对每个 HTML 语句执行追加操作，而是利用反引号或反勾号键（`）执行添加操作并执行单一操作，这可视作是一种优化行为。接下来将利用 jQuery Cycle 添加幻灯片显示功能，稍后将对此加以讨论。

前述内容使得 Web 页面可将全部学生数据加载至 HTML 文件中，下面将利用这些数据构建 Carousel 应用程序。这里将使用 jQuery Cycle 插件在通知板应用程序上旋转学生信息。jQuery Cycle 幻灯片插件支持多种浏览器上的各种渐变效果，例如褪色、展开、擦除、缩放、滚动和混洗等效果。此外，该插件还支持暂停、单击触发器和响应回调等特性。

出于简单考量，当前 Carousel 示例程序仅支持某些基本操作，例如褪色、旋转以及基于鼠标悬停的暂停功能，以使轮播提示板应用程序处于暂停状态，进而显示当前学生的信息。最后，我们还将设置速度和超时值，这将确定学生间的渐变时间，如下所示。

```
<script type="text/javascript">
$(document).ready(() => {
  let htmlContent = ``;
  $.getJSON('http://localhost:3300', (data) => {
    console.log("data", data);
    $.each(data, (index, student) => {
      htmlContent += `<div class="student">
      <div class="col-lg-12 text-center">
        <h3>${student.level} of the Month</h3>

        <h4 class="lead">${student.firstname} ${student.lastname}</h4>

        <p>${student.class}</p>
      </div>
    </div>`;
    });
    $('#students').html(htmlContent);

    $('#students').cycle({
```

```
        "cleartypeNoBg": true,
        "fx": "fade",
        "pause": "1",
        "prev": "#prev",
        "next": "#next",
        "speed": 500,
        "timeout": 10000
    })
  })
})
</script>
```

上述代码片段设置了循环插件，并将其添加至学生的 div 元素中。该循环插件作为参数接收一个 JSON 对象，并将旋转器功能加入 div 元素中。在该 JSON 对象中，我们添加了 4 个属性，即 fx、pause、speed 和 timeout。其中，fx 确定了在 HTML 元素上执行的效果；fade 则是 Cycle 插件的另一种显著效果；jQuery Cycle 插件所支持的其他效果还包括 shuffle、zoom、turndown、scrollRight 和 curtainX。第二个所采用的属性是 pause，当用户悬停于旋转器元素上时，用于确定是否终止旋转行为。相应地，该属性接收一个 true 或 false（0 或 1），进而确定是否暂停旋转行为。接下来的两个属性分别是 speed 和 timeout，用于确定旋转的速度以及当次显示时间。当包含更新后的脚本的 Web 页面载入浏览器中时，全部学生对象将被解析至一个局部 JavaScript 字符串变量中并被追加至 DOM 中。另外，仅旋转器对象中的第一个元素将被显示，其他元素将处于隐藏状态。该功能由 Cycle 插件在后台进行处理。图 6.4 显示了上述示例代码生成的 Carousel 程序结果。

**Freshman of the Month**

**John Doe**

Environmental Toxicology

图　6.4

下面将进一步丰富页面的用户体验，即添加一个处理程序和一个用户自定义控制器，以处理旋转功能，如下所示。

```
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript" src="jquery-3.2.1.min.js"></script>
  <script type="text/javascript" src="jquery.cycle.js"></script>\
  <script type="text/javascript">
  $(document).ready(()=>{
    let htmlContent = ``;
    $.getJSON('http://localhost:3300', (data)=>{
      console.log("data", data);
      $.each(data, (index, student)=>{
        htmlContent+=`<div class="student">
          <h3>${student.level} of the Month</h3>`;
        htmlContent +=`<h4>${student.firstname} ${student.lastname}</h4>`;
        htmlContent+=`<p>${student.class}</p>`;
        htmlContent+=`</div>`;
      });
      $('#students').html(htmlContent);
      $('#students').cycle({
        "cleartypeNoBg": true,
        "fx": "fade",
        "pause": "1",
        "prev": "#prev",
        "next": "#next",
        "speed": 500,
        "timeout": 10000
      })
    })
  })
  </script>
</head>
<body>
  <a href="#" id="prev">Prev</a>
  <a href="#" id="next">Next</a>
  <div id="students"></div>
</body>
</html>
```

在上述代码片段中，我们加入了两个锚元素，对应 ID 值分别为 prev 和 next，并在 cycle 对象中被引用。

cycle 对象中加入了两个名为 prev 和 next 的新属性，对应值表示为 DOM 上的元素的 HTML ID 属性

图 6.5 中的 Prev 和 Next 链接将处理通知板的旋转行为。

Prev Next

**expert of the Month**

**Scott tiger**

Physics

图　6.5

上述内容是一种快速构建由 jQuery 和 JSON 支持的 Carousel 应用程序的方法。此外，该示例还可用于构建更复杂的 Carousel 应用程序，同时分别包含用于照片库和视频库 Carousel 应用程序的图像和视频内容。

## 6.3　Bootstrap 简介

目前，Carousel 应用程序工作正常，唯一的不足之处是应用程序看起来过于“简陋”。针对这一问题，借助于 Bootstrap，可进一步丰富应用程序的观感。

“Bootstrap 是一个免费的开源前端 Web 框架，用于设计网站和 Web 应用程序”。

——en.wikipedia.org

完成应用程序设计或 GUI 部分所需的所有 UI 组件，如选项卡、模式、列表等，都已经包含在 Bootstrap 中。据此，开发人员可调整某些元素、修改 CSS 属性，进而构建自定义元素。接下来将在 Carousel 应用程序中实现 Bootstrap。

### 6.3.1　设置 Bootstrap

Bootstrap 提供了一种较为简单的方式设置应用程序库。类似于其他第三方库，我们可直接提供一个内容分发网络（Content Delivery Network，CDN），或者下载对应的文件。

Bootstrap 针对用户界面交互和事件也包含了 JavaScript 链接，但在当前应用程序中，

我们仅需要使用 CSS。

以下链接需要置于 head 标签中。

```
<link rel="stylesheet" type="text/css" href=
"https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/css/
bootstrap.min.css">
```

我们通过上述链接下载应用程序的 CSS 文件，这样，即使用户处于离线状态，或者在缺少互联网的情况下也可对此加以引用。

待下载完毕后，对应文件名为 bootstrap.min.css，并存储于相应的应用程序目录中。这将链接至 HTML 文件，如下所示。

```
<link rel="stylesheet" type="text/css" href="bootstrap.min.css">
```

当检测上述 CSS 文件是否被成功链接时，需要在浏览器中加载当前应用程序。图 6.6 显示了成功的 Bootstrap 样式表链接。

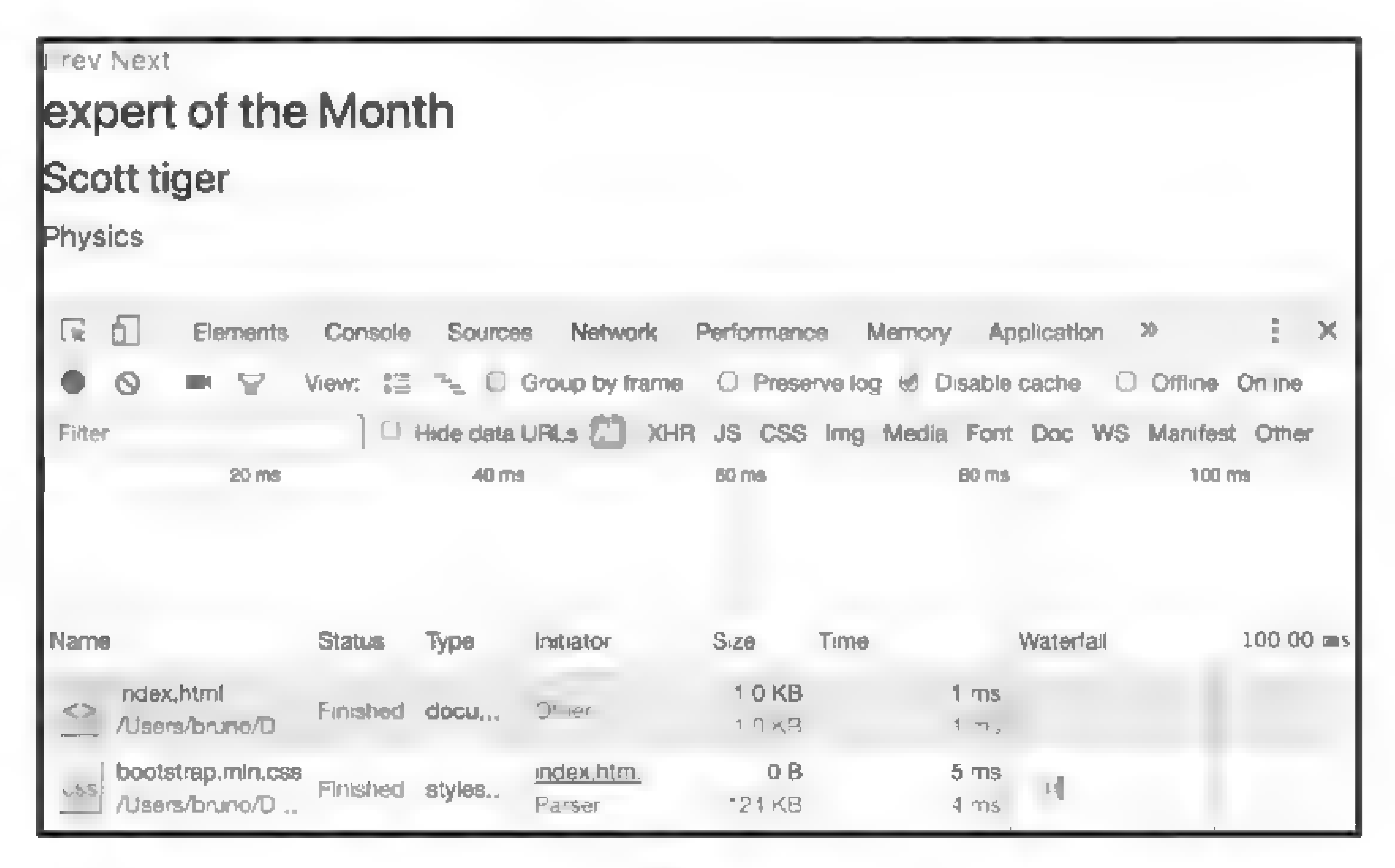



图　6.6

其中，Status 选项卡中包含了一个 Finished 状态，这意味着，CSS 文件已被成功载入。下一步是实现某些布局样式。

## 6.3.2　Bootstrap 响应性和样式

下列各项步骤将引领我们实现应用程序的样式化。此处首先讨论布局元素，随后将重点讨论某些特定元素。

（1）当添加布局样式时，需要设置一个名为 container 的魔术类。该类可针对每种设备处理 Web 应用程序的布局问题，并应添加至所有元素的父元素中。具体来说，当前正在构建的应用程序在 body 元素之后有一个 div 元素，这是所有 Carousel 元素的父元素，如下所示。

```
<div id="students" class="container"></div>
```

最好不要将 body 元素包含在容器中。稍后，它可能会对设计更多的父块造成限制。

在添加了 container 类后，可得到如图 6.7 所示的结果。

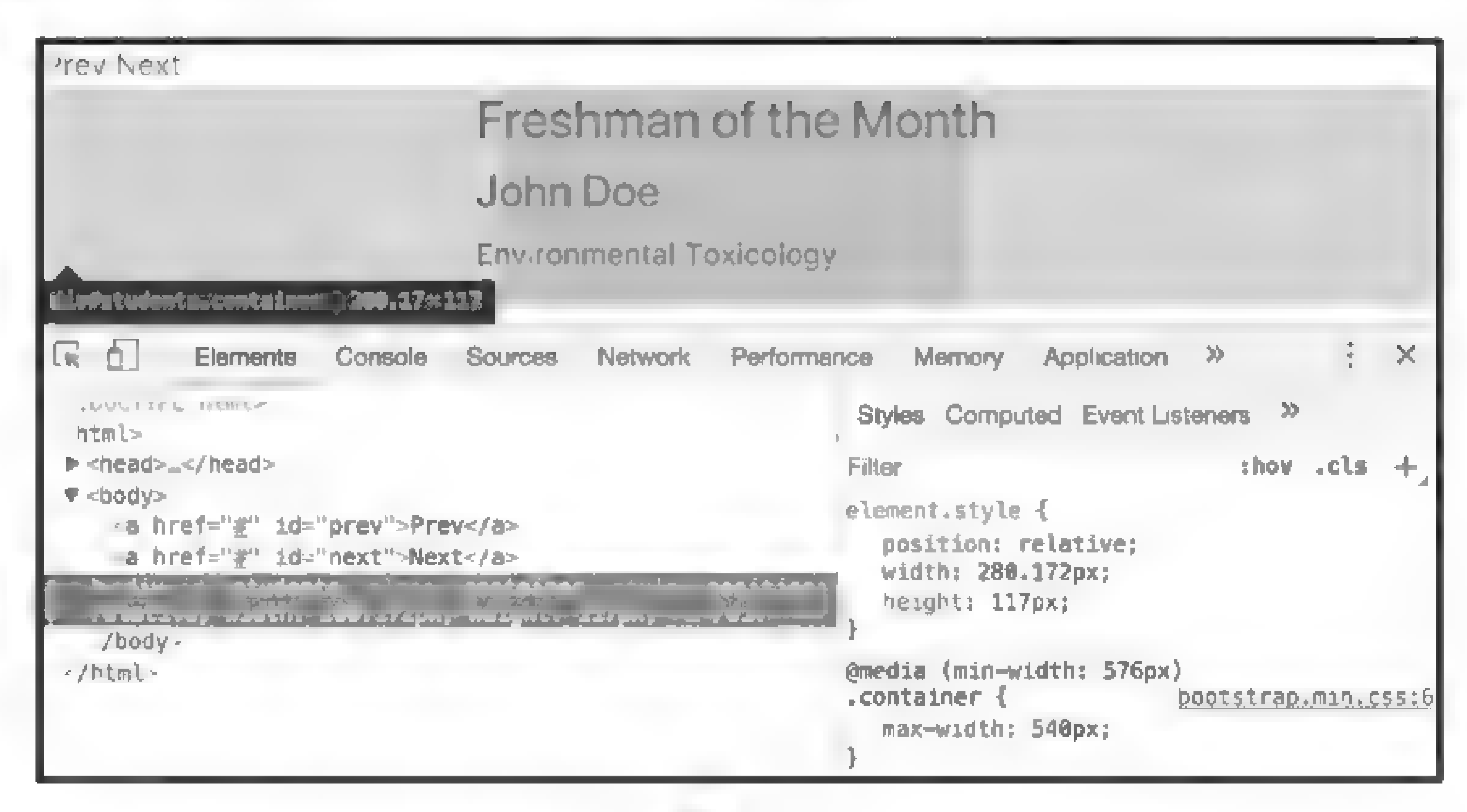



图　6.7

（2）在应用了 container 之后，还需要实现 Bootstrap 的网格系统。Bootstrap 网格系统由一个单行类和多个列类组成。其中，列进一步划分为 12 个子列。在当前示例中，全部内容将添加至 1 行、1 列中。由于内部内容是在父 div 中动态加载的，因而需要在 jQuery 的每个循环回调中进行更改，如下所示。

```
$.each(data, (index, student) => {
  htmlContent += `<div class="student">
    <div class="col-lg-12 text-center">
      <h3>${student.level} of the Month</h3>
      <h4 class="lead">${student.firstname}
      ${student.lastname}</h4>
      <p>${student.class}</p>
    </div>
  </div>`;
});
```

图 6.8 显示了生成的结果。



图　6.8

（3）目前，布局效果看起来较为传统，接下来将关注单个元素，并为以下元素提供 Bootstrap 体验。

- 导航按钮。
- Bootstrap 文本。

综上所述，最终的 HTML 模板如下所示。

```
<!DOCTYPE html>
<html>

<head>
  <link rel="stylesheet" type="text/css" href="bootstrap.min.css">
  <script type="text/javascript" src="jquery-3.2.1.min.js"></script>
  <script type="text/javascript" src="jquery.cycle.js"></script>
  <script type="text/javascript">
  $(document).ready(() => {
    let htmlContent = ``;
    $.getJSON('http://localhost:3300', (data) => {
      console.log("data", data);
      $.each(data, (index, student) => {
        htmlContent += `<div class="student">
        <div class="col-lg-12 text-center">
        <h3>${student.level} of the Month</h3>
        <h4 class="lead">${student.firstname} ${student.lastname}</h4>
        <p>${student.class}</p>
        </div>
        </div>`;
      });
      $('#students').html(htmlContent);
      $('#students').cycle({
        "cleartypeNoBg": true,
```

```
        "fx": "fade",
        "pause": "1",
        "prev": "#prev",
        "next": "#next",
        "speed": 500,
        "timeout": 10000
      })
    })
  })
  </script>
</head>
<style type="text/css">
.btn-left {
  position: absolute;
  left: 20px;
  top: 10%;
}

.btn-right {
  position: absolute;
  right: 20px;
  top: 10%;
}
</style>

<body>
  <a href="#" id="prev" class="btnbtn-primary btn-left">Prev</a>
  <a href="#" id="next" class="btnbtn-primary btn-right">Next</a>
  <div id="students" class="container"></div>
</body>

</html>
```

图 6.9 显示了修改后的模板内容。

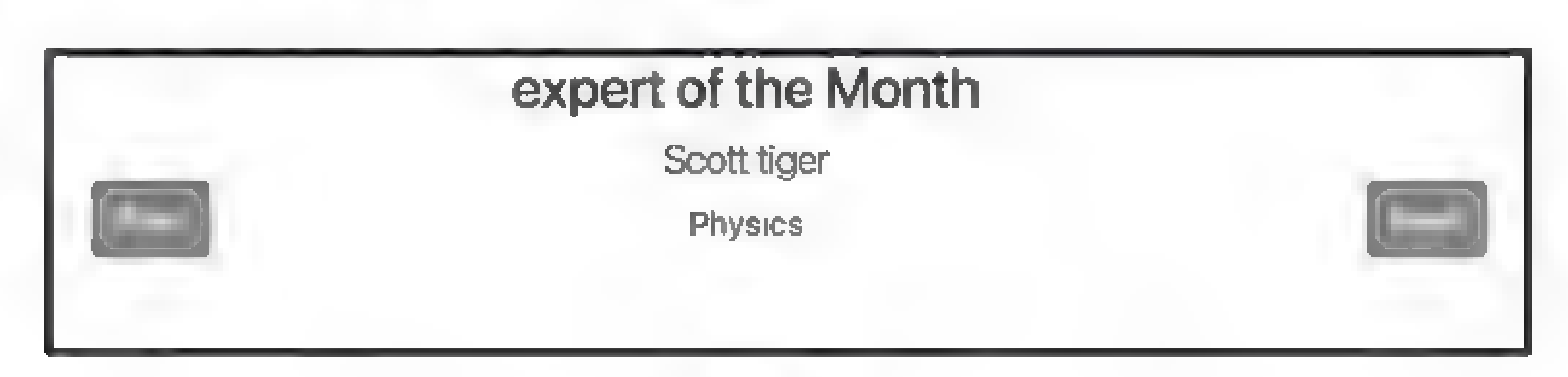



图　6.9

至此，Carousel 应用程序将暂告一段落。我们的 JSON 之旅实现了前端应用程序。在

后续章节中，还将进一步探讨服务器端和客户端上 JSON 数据的替代方案。

## 6.4 本章小结

本章整合了 JavaScript、jQuery 和 JSON，并构建了一个简单的 Carousel 提示板应用程序。其间我们逐步完成了以下各项步骤：摄取数据提要、从该数据提要动态构建一个模板、将数据提要追加到 div 元素，然后将 div 元素绑定到 cycle 插件。从通知板旋转器应用程序中可以看到，即使较大的 Carousel 项目也只需较少的工作即可完成。最后，本章还通过 Bootstrap 对应用程序进行了修正，同时学习了如何在应用程序中实现不同的 UI 构造模块。第 7 章将讨论 JSON 的替代实现方案。

# 第 7 章　JSON 的替代方案

截止到目前，我们仅将 JSON 用作一种 HTTP 数据交换格式，本章将考查一些较为流行的替代方法。最近几年来，在编程和脚本语言中，软件模块和数据包的数量都在急剧增长。本章主要涉及以下 JSON 实现。

- PHP 和 Node.js 中的依赖关系管理。
- 用于在 PHP 和 Angular 中存储应用程序配置的 JSON。
- Angular 中的应用程序元数据。
- Node.js 中的常量。
- 针对嵌入式模板或服务器端显示的 JSON 元数据应用。

诸如 PHP 或 JavaScript 这一类脚本语言涵盖了大量的软件包和模块。这一类预先构建的软件包其优点主要体现在，它们提供了一些即用功能，并且经过了社区的大量测试。另外，当在软件项目中引入单一框架或多个框架时，还需要了解如何将这些框架加载到项目中，如何从当前项目的不同部分访问它们，这些框架是否存在依赖关系，以及它们如何影响整个项目。这一类问题可通过依赖关系管理器加以解决。

## 7.1　依赖关系管理

依赖关系管理器是一种软件程序，它跟踪程序运行所需的所有基本程序。软件开发生命周期中常见的实践是利用单元测试框架执行单元测试；反过来，单元测试框架也需要使用一些基础库；或者通过某些配置启用框架的应用。

上述操作的常见处理方式是编写快速的脚本。但随着项目的不断扩展，依赖关系也随之增长。同样，跟踪这些变化，并确保在项目上工作的不同团队获得相关更新（由脚本完成）将变为一项艰巨的任务。通过引入依赖关系管理器，可确保整合过程的自动化处理，在增加了一致性的同时还可节省大量的时间。

### 7.1.1　在 PHP 中使用 composer.json

依赖关系的管理过程相对复杂，对于刚开始向项目中添加新框架、设置项目并使其运行的新晋开发人员来说，这是一项艰巨的任务。依赖项管理器（例如 PHP 的 Composer）可以解决这个问题。它被认为是“所有项目之间的黏合剂”，这是有原因的。Composer 使

用 JSON 跟踪既定项目的全部依赖关系。Composer 的主要工作是从远程位置下载库，并采用本地方式对其进行存储。当告知 Composer 我们需要哪些库时，可配置 composer.json 文件。该文件记录了所有的库、版本以及库的部署环境。例如，单元测试框架库不可应用于生产环境中。假设某人正在对产品实例进行随机测试，但通过运行单元测试删除了整个用户列表。对此，必须从之前的数据库备份中恢复整个用户表。

图 7.1 显示了如何利用 JSON 处理依赖关系管理。




图　7.1

在 composer.json 文件中，我们加入了两项内容，进而安装 PHP 和 PHPUnit 的特定版本。当该文件被添加至当前项目后，即可采用 Composer 的 install 命令安装这些依赖关系。此外，Composer 还包含了一个 update 命令，并关注相关数据包的更新状态。

关于 Composer 的更多信息，读者可访问 http://www.getcomposer.org。

## 7.1.2　基于 package.json 的 Node.js

Node.js 是一个流行的软件平台，并采用 JSON 数据格式跟踪依赖关系。Node 包管理器（Node Packaged Modules，NPM）可供开发人员将外部模块安装、集成至其代码中。对于每个 Node.js 项目，在根文档中存在一个 package.json 文件，并可跟踪所有的元数据，例如项目名称、作者名称、版本号、运行项目所需的模块，以及运行项目所需的底层守护进程或引擎。图 7.2 显示了 Node.js 项目中的一个 package.json 示例文件。

package.json 文件是一个较大的 JSON 对象，用于跟踪元数据，例如项目名称、作者的详细信息以及所需的模块。

关于 NPM 的更多信息，读者可访问 https://www.npmjs.org



图　7.2

# 7.2　存储应用程序配置的 JSON

在 JSON 之前，配置内容一般存储于文本文件或特定的语言文件中，例如 config.php（PHP 语言）、config.py（Python 语言）或 config.js（JavaScript 语言）。所有这些文件均可被与语言无关的 config.json 文件加以替换，同时针对非 JavaScript 库采用 JSON 库对其进行解析。

## 7.2.1　PHP 和 Python 中的配置

图 7.3 显示了 config.json 文件。

在 config.json 文件中，元数据存储为 JSON 对象。这里，我们可指定一些较为重要的信息，例如项目名称、项目环境（根据文件所处的服务器而变化）、需要在应用程序引导过程中自动加载的类，以及需要排除的类或文件夹。最后，通过 RECURSIVE 键，还可进一步指定空文件夹和包含文件的文件夹。

引导是指应用程序的启动过程，其间，应用程序将处于就绪状态并实现其相应的功能。

当 config.json 文件配置完毕后，可在 Python 中使用 json.loads()方法，或者在 PHP 中使用 json_decode()方法，进而解析 config 对象以检索数据。此外，JSON 对象还可用于存




图　7.3

储元数据模式，若团队中的某位成员修改了数据库，这将有助于其他成员更新其数据库模式。对此，一种较为明智的作法是在 schema.json 文件中编写一个触发器，如果该文件被更新，则数据库中的对应模式也需要被更新，并通过数据库迁移脚本反映新的更改内容。图 7.4 显示了 schema.json 文件。




图　7.4

在 schema.json 示例中，我们构建了一个 JSON 模式对象，以存储数据库模式信息。其中，client 表示为当前模式中的表名。client 表包含了 3 列，分别是 ID、名称和客户状态。也就是说，对应客户处于启用状态或禁用状态。另外，每列中包含了提供模式信息的 JSON 列对象，例如数据类型、列大小，以及是否包含默认值或主键约束条件。

## 7.2.2　在 Angular 5 中进行配置

最新的 Angular 框架是一个前端开发工具集；除此之外，还利用客户端的各种测试库提供了端到端的测试环境。Angular 5 之前称作 angular.js，作为一个 JavaScript 库，它主要针对基于浏览器的单页应用程序提供了模型-视图-控制器架构。目前，Angular 通过谷歌予以维护。

在最新的 Angular（版本 2 以上）中，我们将采用 TypeScript 作为基本的开发脚本语言。根据 TypeScript 文档中描述，TypeScript 通过类型严格的特性扩展了 JavaScript，并向代码中添加了更多的语法糖。此外，对于部署和浏览器的使用来说，还可以将代码转换为纯 JavaScript。转换过程将把 TypeScript 编译为 JavaScript。

为了进一步理解 Angular 中的配置特性，需要在机器中配置该框架。这里，建议读者遵循以下链接中提供的各项步骤配置 Angular 框架：https://angular.io/guide/quickstart#devenv。

最终，对应结果如图 7.5 所示。




图　7.5

my-app 的上述结构是通过 Microsoft 的 vscode 编辑器实现项目化结构的。对此，读者可访问 https://code.visualstudio.com/以了解更多信息。

所有 JSON 文件都是在 Angular 中构建应用程序所需的配置文件，下面将对此进行逐一讨论。

### 1. 基于 tslint.json 的 linting

linting 是检查、调试代码语法和样式的过程，这对于代码的维护和可读性来说十分重要。另外，linting 还可以消除一些未来可能导致 bug 的错误。tslint.json 利用 JSON 规范结构供开发人员编写特定的项目规则。读者可能已经注意到，linting 配置与 TypeScript 相关——此处前缀为 ts。

以下示例展示了 tslint.json 文件中 my-app 的一些 linting 规则，其内容具有自解释特征。另外，读者还可参考代码中的注释内容以对其进一步了解。

```
{
 "rules":
 {
 //short hand arrow return: getData((someVariable)=>someVariable)
  "arrow-return-shorthand": true,
 //Indentation of spaces allowed
  "indent": [
    true,
    "spaces"
  ],
 //Semicolon after every end of statement neccessary
  "semicolon": [
    true,
    "always"
  ],
 //Conditional operator used for type match: allowed
  "triple-equals": [
    true,
    "allow-null-check"
  ]
 }
}
```

### 2. 利用 tsconfig.json 配置 TypeScript

tsconfig 也称作 TypeScript 项目的根文件，负责初始化 TypeScript 项目，并在编译 ts 时使用特定的配置规则。下面考查一些较为重要的规则及其具体含义。

```
{
//On saving the ts file, compile the ts to js
  "compileOnSave": true,
  "compilerOptions": {
//Targeting option for ecmascript versions
    "target": "es5",
//Library files that need to be included for target
    "lib": [
      "es2017",
      "dom"
    ]
  }
}
```

全局 tsconfig 文件使用特定源目录的 tsconfig.json 中的 extend 属性实现继承。

关于键及其应用，读者可访问 http://json.schemastore.org/tsconfig 以了解更多信息。

### 3. 使用 package.json 和 package-lock.json 文件

前述内容曾介绍了 package.json，它主要用于维护与应用程序相关的信息，例如名称、作者、测试脚本等，并管理产品和开发级别所需的依赖模块。最近，NPM 发布了新的功能，并在数据包更新或安装过程中生成 package-lock.json 文件。

图 7.6 显示了 package-lock.json 文件。




图　7.6

package-lock.json 文件可指定基于时间更新的、所有的数据包记录，且不必逐个访问应用程序中的每个节点模块，以检查其依赖性信息。此外，package-lock.json 文件还提供了模块的树形结构，并在节点模块安装过程中提供了性能方面的优势。

#### 4. 使用 angular-cli.json 文件

最新的 Angular 针对多项功能提供了命令行界面（cli），例如应用程序启动、部署或创建组件。相应地，用于执行此类任务的命令一般采用 ng 命令前缀。例如，ng serve 命令负责编译代码，并通过 webpack 同步监视文件更改。

*webpack 是一个开源工具，并可针对 JavaScript 应用程序生成特定的环境，读者可访问 https://webpack. js.org/以了解更多信息*

在应用程序级别上执行的全部操作均可在 angular-cli.json 文件中进行配置。图 7.7 显示了安装过程中提供的默认配置信息。

```
.angular-cli.json
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "project": {
    "name": "my-app"
  },
  "apps": [
    {
      "root": "src",
      "outDir": "dist",
      "assets": [
        "assets",
        "favicon.ico"
      ],
      "index": "index.html",
      "main": "main.ts",
      "polyfills": "polyfills.ts",
      "test": "test.ts",
      "tsconfig": "tsconfig.app.json",
      "testTsconfig": "tsconfig.spec.json",
      "prefix": "app",
      "styles": [
        "styles.css"
      ],
      "scripts": [],
      "environmentSource": "environments/environment.ts",
      "environments": {
        "dev": "environments/environment.ts",
        "prod": "environments/environment.prod.ts"
```

图 7.7

需要注意的是，angular-cli.json 中的 apps 键由一个数组构成，这意味着，可管理 Angular 框架中多个数据源的多个 App。下面利用 ng serve --app，或简单地采用 ng serve 运行应用程序，进而对设置项进行测试，对应输出结果如图 7.8 所示。

```
Last login: Wed Apr 11 17:14:12 on ttys010
brunos-MacBook-Pro:my-app bruno$ ng serve --app 0
** NG Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200
**
Date: 2018-04-11T16:37:16.281Z
Hash: fd7734d6620b30a52717
Time: 9010ms
chunk {inline} inline.bundle.js, inline.bundle.js.map (inline) 5.83 kB [entry] [rendered]
chunk {main} main.bundle.js, main.bundle.js.map (main) 6.03 kB {vendor} [initial] [rendered]
chunk {polyfills} polyfills.bundle.js, polyfills.bundle.js.map (polyfills) 199 kB {inline} [initial] [ren
dered]
chunk {styles} styles.bundle.js, styles.bundle.js.map (styles) 11.3 kB {inline} [initial] [rendered]
chunk {vendor} vendor.bundle.js, vendor.bundle.js.map (vendor) 2.29 MB [initial] [rendered]

webpack: Compiled successfully.
```

图　7.8

图 7.8 中的输出结果表明代码被成功编译。

通过运行 ng built 命令，将使用预先（Ahead of Time，AOT）编译器编译代码，并创建应用程序的部署构建，考查如图 7.9 所示的结果。




图　7.9

注意，在处理过程完毕后，将生成一个 dist 目录，表示转换为 JavaScript 的可部署代码。相关原因可描述为：如果修改 angular-cli.json 文件（关键数据"outDir": "dist"变为"outDir": "distribution"），那么还会将构造目录名修改为 distribution。

这也是新 Angular 的突出特性之一，它充分利用了 JSON 的可配置特性。关于 Angular 配置中其他键工作方式的更多细节，Angular 背后的团队已经以事例的形式汇总了 CLI 模块文档。读者可访问 https://github.com/angular/angular-cli/wiki/stories 以了解更多信息。

## 7.3　存储应用程序元数据的 JSON

在依赖管理器中，JSON 还可用于存储软件项目的元数据。那么，元数据与之前解释的应用程序配置有何不同？元数据有别于以下事实：配置是应用程序正常工作所需的一组定制的设置项。因此，可以将配置数据称为元数据的一种类型或子集。为了在整体上理解这一概念，下面考查 JSON 作为元数据的不同实现。

### 7.3.1　Angular 5 中的元数据

在 Angular 中，元数据是在处理某个特定功能的类时加以使用的。元数据对类进行配置，以使将其用作组件或服务。另外，元数据是通过类装饰器或属性元数据实现的。稍后将学习类装饰器，因为它适用于 JSON 基本上下文。

在 Angular（版本 2 以上）中，首先需要讨论的是 NgModule。

“NgModule 是一个带有@NgModule 装饰器函数的类，它接受一个元数据对象，并通知 Angular 如何编译代码”。

——angular.io

每个类可以附带一个装饰器。对于 NgModule，对应类已经在 Angular 核心库中加以定义。我们需要使用 NgModule 类的装饰器识别所有组件、供应者（也称作服务），以及开发人员创建的或 Angular Core 导入的模块。

接下来考查以下源自 app.modules.ts 文件中的代码。

```
@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    AppComponent
  ],
  providers: [],
  bootstrap: [
    AppComponent
  ]
})
```

在上述代码中，NgModule 类可以像函数一样调用，并使用 JavaScript 对象作为参数。注意，对此需要通过注解操作符“@”向 NgModule 关键字添加前缀，即装饰器的语法

声明。

一个稍显复杂的例子是 Angular Core 中的 Component 类。在 app.component.ts 文件中，包含了以下代码片段。

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';
}
```

这里使用了@Component 装饰器装饰 AppComponent 类。这意味着自定义类 AppComponent 绑定到@Component 装饰器。组件元数据用于指定以下内容。

- ❑ selector 组件的名称，用作一个 HTML 标签加载特定的组件。
- ❑ 需要加载到 selector 标签附近的模板。
- ❑ CSS 或样式表文件。

注意，元数据并非总是可选的，它也可以是强制性的。在前面的例子中，对于 Component 类来说，传递一个选择器并指定一个模板是 Angular 在 UI 中加载组件所必需的。

这也是 JSON 元数据这一概念在 Angular 框架中的实现方式。接下来将考查 Node.js。前述内容简要介绍了的 Node.js 以及 package.json 的含义，下面将在 Node.js 应用程序中使用常量 JSON 数据，并通过常量构建一个简单的 Node.js 示例服务器。

## 7.3.2　Node.js 中的常量

在 Node.js 中，可将元数据定义为常量，并在模块间的应用程序中对其加以使用。下列代码片段用于启用 Node.js 中的基础服务器。

```
const http = require('http');
const constants = require('./constants');
const port = constants.port;
http.createServer((req, res) => {
  res.end(`Hello ${constants.audience}`);
}).listen(port);
console.log(`Node Server is running on port: ${port}`)
```

前述章节中曾讨论了 Node 服务器，对应代码创建了一个 HTTP 服务器，并监听本地指定端口上的传入请求。与第 6 章中的示例不同，此处导入了一个 constants 模块。

constants.js 模块涵盖了下列代码块。

```
module.exports = {
  "port": 3300,
  "audience": "readers"
}
```

上述代码由导出的 JSON 构成。全部键均为常量，并在其他模块间用作单例。注意，仅当加载了节点进程，且不执行永久性地写入操作时，这些键才是可变的。

在 Node.js 中，此类常量文件可以包含启动应用程序所需的任何可配置数据，或者是跨模块、全局使用的数据。更具体地说，port 键是一个配置类型的元数据键，audience 可称为常量类型的元数据。

### 7.3.3　模板嵌入机制

嵌入模板是将数据模型插入至视图的过程。这里的模型可以是 JSON 或简单的字符串。模板嵌入过程中也产生了新的前端技术，例如 react.js、ember.js、handlebars.js 等。采用此类模板引擎，可在 Node.js 服务器自身中开发 Web 客户端。这也使得 JavaScript 变为一种同构技术。下面将利用 handlerbar.js 在服务器端实现此类机制。

handlebar.js 是一个简单的模板引擎，并可将 JSON 数据嵌入模板中。根据以下各项步骤，我们将在 Node 服务器上对其加以实现。

（1）首先，使用下列命令并作为节点模块安装 handlebar。

```
npm install --save handlebars
```

（2）待安装完毕后，即可在 app.js 中包含 handlebar 模块，如下所示。

```
const handlebar = require('handlebars');
```

（3）在与 handlebar 模块协同工作之前，首先生成一个通过响应发送的 HTML。下列 index.html.js 文件包含了模板数据。

```
module.exports = `
<!DOCTYPE html>
<html>
<head>
  <title>Hello readers</title>
</head>
<body>
  <h2>Greeting audience!</h2>
  <h3>Life is beautiful</h3>
```

```
</body>
</html>
```

此处在考虑 HTML 数据时遵循了相应的命名规则（虽然并不标准），并通过 js 扩展导出为节点模块。

（4）针对 HTML 响应生成一个路径/html。下列代码表示作为响应发送 HTML 时所做的一些修改。

```
const http = require('http');
const constants = require('./constants');
const template = require('./index.html');
const port = constants.port;
const handlebar = require('handlebars');
//console.log("template",);
http.createServer((req, res) => {

  if(req.url == '/html'){
    res.setHeader('content-type', 'text/html');
    res.end(template);
  }else
    res.end(`Hello ${constants.audience}`);
}).listen(port);
console.log(`Node Server is running on port: ${port}`)
```

上述代码导入了模板文件 require('./index.html')，并于随后将响应头设置为 text/html。这样，客户端浏览器就知道如何解释从服务器接收到的响应。

（5）利用 http://localhost:3300/html 请求节点服务器，对应的输出结果如图 7.10 所示。

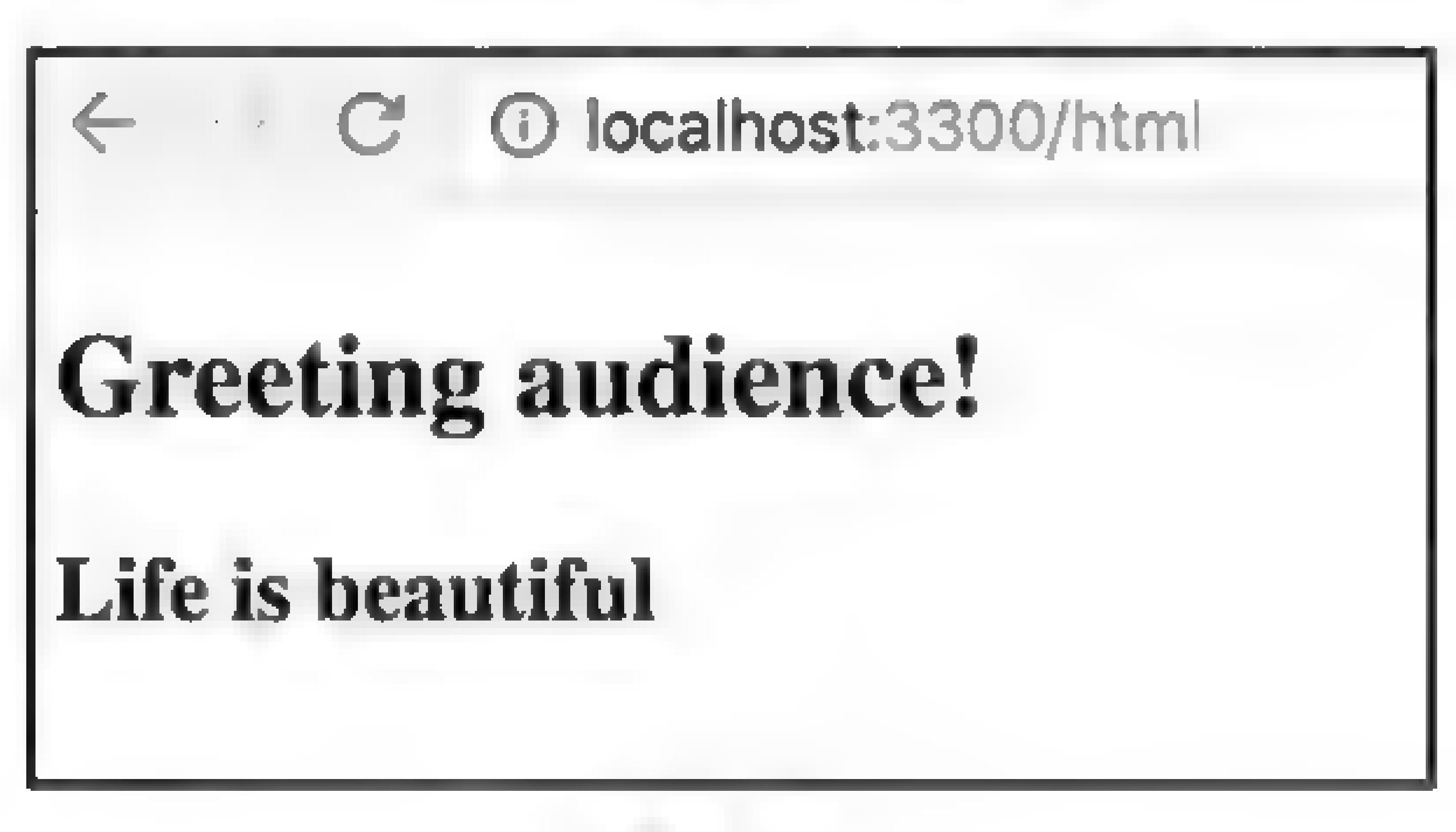



图　7.10

（6）利用不同的数据集嵌入同一模板。出于演示目的，此处将 audience 修改为 readers，并将 beautiful 修改为 simple。对此，需要在 HTML 中实现两个占位符，如下所示。

```
//index.html.js
module.exports = `
<!DOCTYPE html>
<html>
<head>
   <title>Hello readers</title>
</head>
<body>
  <h2>Greeting {{audience}}!</h2>
  <h3>Life is {{adjective}}</h3>
</body>
</html>
```

（7）创建一个包含键 audience 和 adjective 的模型，如下所示。

```
{
  "audience": "Readers",
  "adjective": "simple"
}
```

（8）当嵌入新数据（JSON）时，须将代码更新为如下所示。

```
const templateData = handlebar.compile(template)({
  "audience": "Readers",
  "adjective": "simple"
})
```

作为属性，handlebar 包含了一个 compile()方法，该方法将 JSON 数据嵌入 HTML 中。最后，不要忘记将响应数据从 template 更改为 templateData，如下所示。

```
res.end(templateData);
```

（9）重启节点服务器并访问 localhost:3300/html，浏览器中所显示的输出结果如图 7.11 所示。




图　7.11

需要注意的是，文本 beautiful 被替换为 simple。当前原型相对简单，根据需要，还可

以通过数组、对象等来呈现模板数据，进而向模板数据中增加一些复杂性。handlebar 是一个较为稳定的库，并可对此进行适当处理。关于 handlebar 库，读者可访问 http://handlebarsjs.com/以了解更多信息。

## 7.4　与 YAML 进行比较

YAML 是另一种与软件语言无关的数据交换格式，这种格式正逐渐流行起来。YAML 是 YAML Ain't Markup Language 的递归缩写，通常用于存储配置、模式和属性等元数据。YAML 被认为是一种人类可读的数据序列化标准，它依赖于行结束符的空格、位置和简单字符，类似于 Ruby 和 Python 等流行脚本语言。YAML 特别关注元素之间的间距，并且不支持制表符。与 JSON 类似，YAML 键/值对由冒号分隔。与文本格式类似，连字符用于指示列表项，而 JSON 则将列表项置于数组或子对象中。由于 YAML 与软件语言无关，因此我们需要解析器来理解该文件中的内容。此类解析器适用于大多数流行的语言，如 PHP、Python、C++、Ruby 和 JavaScript。下面在 YAML 中构建一个 config.json 文件以理解 YAML 的具体含义，如图 7.12 所示。




图　7.12

类似于 JSON 对象，YAML 文件中包含了全部数据。不同之处在于如何将数据作为数据项列表进行排列，以及如何使用间距和位置来排列数据列表。对此，互联网上存在大量的 YAML 资源可用来验证、序列化和反序列化 YAML 数据。

关于 YAML，读者可访问 http://www.yaml.org 以了解更多信息，相关内容将以 YAML 格式予以展示。

## 7.5 本章小结

JSON 正在快速演变为互联网上最为流行的数据交换格式，但并不仅限于数据交换。针对依赖项管理器、包管理器、配置管理器和元数据存储，本章讨论了 JSON 的元数据存储应用方案。除此之外，本章还介绍了 YAML，它被认为是 JSON 的替代方案。第 8 章将考查用于调试、验证和格式化 JSON 的各种资源。

# 第 8 章　hapi.js 简介

在过去的几年里，Node 社区处于较快的发展阶段，并涌现出了各种框架。这些框架实现了多方面的基准测试，例如代码复杂性、体系结构模式、性能以及社区应用。一些框架，例如 Express 和 Koa，以其极简的特性而闻名；而其他框架，如 hapi 和 sail，则提供了可配置的或结构化的编码技术。

本章将讨论 hapi.js，它是一种可配置的框架，并使用 JavaScript 对象作为配置数据。hapi.js 并不是纯粹的 JSON，而是作为 JavaScript 文本提供的典型的对象表示法。本章主要讨论 JSON 的实现和可扩展性，以便采用 hapi 框架构建 JavaScript 服务器。

本章主要涉及以下主题。

- 利用 JSON 实现基本的服务器配置。
- 利用 JSON 配置 API。
- 使用 JSON 元数据和常量。
- 利用 POSTMAN 测试 hapi 服务器 API。
- POSTMAN 下的 JSON。

下面将逐一对此加以讨论。

## 8.1　利用 JSON 实现基本的服务器配置

前述内容在讨论模板嵌入时曾搭建了一个 Node 服务器，我们可在此基础上进行修改，或者生成一个新的 App。如果打算使用相同的 App，则需要完成以下工作。

- 修改 app.js。
- 消除模板相关的实现。

在修改完毕后后，图 8.1 显示了代码库中当前的结构。

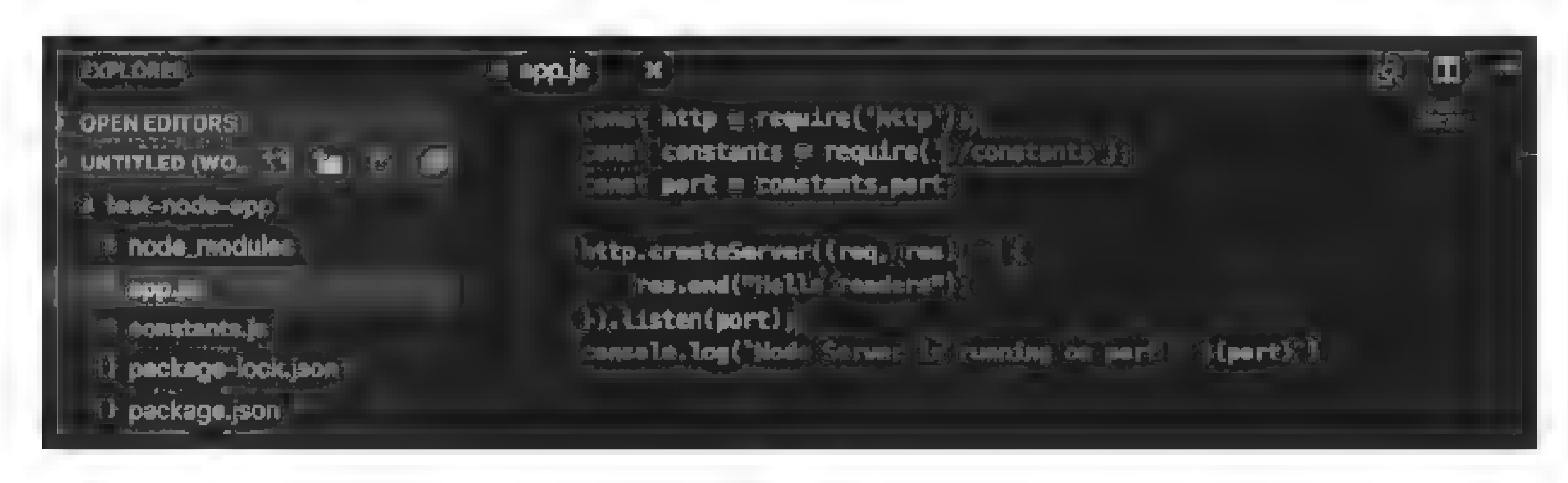



图　8.1

简单地讲，上述代码创建了一个简单的 HTTP 服务器。createServer()方法包含了请求参数 req，用于处理输入请求，以及服务器间的响应实例。此处使用了响应对象的 end()方法，并将响应数据（Hello readers）发送至客户端。

相关步骤如下。

（1）首先需要安装 hapi 框架，对应版本不应低于 v17.x.x。默认状态下，包管理器将为我们安装最新的版本。另外，确保在安装完毕后在 package.json 中对其进行验证。对应的安装命令如下所示。

```
npm install hapi --save
```

Node 包管理器在 App 的节点模块中安装了名为 hapi 的新的依赖关系。

（2）接下来需要在 app.js 中包含 hapi 模块，如下所示。

```
const Hapi = require('hapi');
```

包含 hapi 模块可使我们使用 hapi 框架中的全部特性。

（3）构建一个基本的 hapi 服务器，对应代码如下所示。

```
const Hapi = require('hapi');
const constants = require('./constants');
const port = constants.port;
const server = new Hapi.Server({
  port
});
server.start();
console.log(`Server running at: ${server.info.uri}`);
```

上述代码执行了以下操作。

- ❑ 类似于其他节点模块，我们需要使用一个 hapi 模块。
- ❑ 之前设置了一个自定义常量模块，进而提供一些配置数据。当前，我们仅需使用来自常量的端口。
- ❑ 利用 server()方法创建了一个新的 hapi 服务器实例。注意，针对该服务器，我们使用 JSON 作为配置数据。
- ❑ 最后，使用服务器实例的 start()方法初始化服务器，该方法可视为一个触发器进而启动特定的服务器。

考查以下场景，假设开发团队计划处理两个服务器实例，利用相同的配置，服务器的初始化可根据相关条件轻松地切换。这可通过下列伪代码予以展示。

```
If(server1 is configured)
  Server1.start()
```

```
else
  server2.start()
```

或者以下列并行方式执行。

```
Server1.start();
Server2.start();
```

两个实例可在同一端口上运行

- 执行 node app.js 以初始化服务器。对应结果如图 8.2 所示。

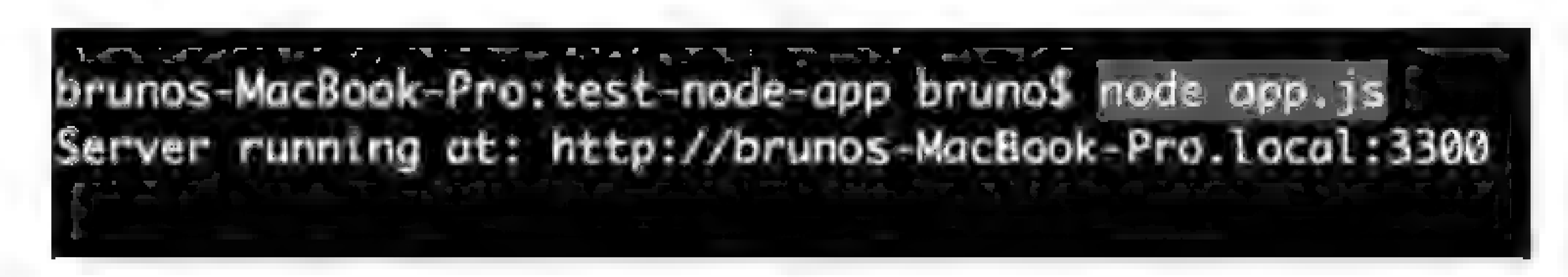



图　8.2

随后，服务器将在 http://localhost:3300 上启动。

在上述输出结果中，不要与 URL 中提及的机器的配置文件名混淆，它仅充当于本地主机。如果将上述地址置于浏览器中，将会收到一个错误代码，表示没有找到错误消息，其原因在于，当前尚未配置任何 API URL 端点。稍后将对此加以讨论。在考查 API 之前，下面首先讨论在 package.json 中配置启动命令，这也是采用 JSON 进行配置时需要注意的另一个特性。

## 8.2　使用 JSON 元数据和常量

如前所述，使用 node app.js 命令可启动服务器，这一点我们已有所了解，但目前尚未讲述如何使用该命令。Node 包管理器（Node Package Manager，NPM）提供了一种方法，并利用 package.json 中的 scripts 键配置服务器的 start 命令，如下所示。

```
{
  "name": "test-node-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" &amp;amp;&amp;amp; exit 1",
    "start": "node app.js"
  },
  "author": "",
```

```
  "license": "ISC",
  "dependencies": {
    "handlebars": "^4.0.11",
    "hapi": "^17.2.0"
  }
}
```

上述代码在脚本的文本中添加了一个 start 键，并作为值提供了实际的 Node 服务器启动命令。新命令的运行方式如下所示。

```
npm start
```

当前，服务器处于运行状态。此处，start 键已被 npm 作为启动服务器的默认命令输入而识别。但该过程并非是真正的自定义操作，我们还可将该命令命名为 new-start，以实现进一步的定制行为。对此，需要在 package.json 中进行如下修改。

```
{
  "name": "test-node-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" &amp;amp;&amp;amp; exit 1",
    "start": "node app.js",
    "new-start": "node app.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "handlebars": "^4.0.11",
    "hapi": "^17.2.0"
  }
}
```

下面利用自定义的启动命令运行服务器，如下所示。

```
npm run new-start
```

npm 解析 package.json，并发现了 new-start 键及其对应的执行命令。此类启动命令并不是 hapi.js 框架所特有的，但却与 npm 框架中的 package.json 相关。此外，还可将其用于 Express、Sail、Koa 或任何其他框架中。

针对各种场景，这仅是 JSON 实现的开始阶段。hapi 服务器与配置代码库相关，接下来将讨论如何利用 JavaScript 对象表示法配置路由。

## 8.3 利用 JSON 配置 API

简而言之，API 是指访问应用程序数据的一种方式，并将其以期望的格式呈现出来。API 实现通过应用程序请求授予对其数据的访问权，这可以是一个 HTTP 请求或 FTP。本书则主要考查 HTTP 请求。

请求基本上由以下内容构成。

- ❑ 统一资源定位符（Unique Resource Location，URL）。
- ❑ 请求头。
- ❑ 请求体。

API 服务器提供了一个端点 URL，以便可通过客户端（如浏览器）访问所需的信息。前述章节曾讨论了如何用 Node.js 实现一个 API 服务器。

下面考查 hapi 如何在 Node.js 上实现此类 API。对此，考查下列代码。

```
const Hapi = require('hapi');
const constants = require('./constants');
const port = constants.port;

const server = new Hapi.Server({
  port
});

server.route({
  method: 'GET',
  path:'/greetings',
  handler(request, h) {
    return 'hello readers!';
  }
});
server.start();
console.log(`Server running at: ${server.info.uri}`);
```

在具体讨论代码之前，首先在浏览器中访问 http://localhost:3300/greetings，进而可得到输出结果 Hello readers!。

上述代码中引入了 hapi 服务器实例中的新方法 route()。函数路由所需的最小配置包含方法、路径和处理程序，具体如下。

- ❑ 方法：表示为 HTTP 资源方法，并以此访问 HTTP 服务器上的资源。其他基本方法还包括 GET、POST、PUT 和 DELETE。在当前示例中，我们将使用 GET 方法。

- ❑ 路径：表示为一个 URL 或端点。这些端点与域连接工作，以定位、请求来自任何客户端的特定数据，如浏览器。在当前示例中，我们将使用 http://localhost:3300/greetings。
- ❑ 处理程序：每个 URL 路径均包含一个特定的目标；或提供与其方法相关的某种功能。此类功能是使用处理程序的函数生成的。在 Hapi v17.2.0 中，处理程序键包含一个带有两个参数的回调函数，如下所示。
  - 第一个参数表示所接收的实例请求。其中，请求实例由所有的请求信息构成，例如请求头、URL 参数、请求体、事件、原始节点请求实例等。
  - 第二个参数则是 hapi 服务器实例和我们所提供的路由配置数据上下文的组合。

数据在 handler 回调中返回，可以是简单的字符串，也可以是由 hapi 转换成的服务器响应，甚至是 HTML 的复杂的 JSON。此类数据类型的设置头（例如内容类型）将通过 hapi 自身予以处理。

下面将路由功能隔离到特定的路由文件中，具体步骤如下。

（1）从单独的文件（例如 routes.js 文件）导出路由配置对象，如下所示。

```
//routes.js
module.exports = [{
  method: 'GET',
  path:'/greetings',
  handler(request, h) {
    return `<b>hello readers!</b>`;
  }
}]
```

注意，我们已经导出了一个数组，因此可以传递一个数组中的多个路由配置对象。

（2）移除位于 app.js 中的路由配置对象，并在 app.js 文件中包含 routes.js，如下所示。

```
const Hapi = require('hapi');
const constants = require('./constants');
const routes = require('./routes');
const port = constants.port;
const server = new Hapi.Server({
  port
});
server.route(routes);
server.start();
console.log(`Server running at: ${server.info.uri}`);
```

需要注意的是，此处作为参数向 server.route()方法中传递了 routes 常量。在终端中重新启动 hapi 服务器，将会得到相应的输出结果。这里，将路由从 App 中隔离出来可视为

一种较好的方法，在提升了路由控制器的可维护性之外，还保持了 app.js 的简洁性。

（3）重新启动 hapi 服务器，在浏览器中访问 http://localhost:3300/greetings 以验证是否可正常工作。

## 8.4 在 hapi 中配置插件

插件提供了一种方式，可在不同的代码部分中处理业务逻辑。插件的实现不同于中间件，以及用于特定目的的任何第三方工具方法。

下面创建一个插件以了解插件的工作方式。当集成插件时，需要在应用程序中生成一个名为 plugins.js 的文件。该文件由下列代码片段构成。

```
exports.logRequest = {
  register(server, options){
    console.log("A plugin got called!");
  },
  name: "logRequest"
}
```

上述代码包含了构成插件所需的最少属性。其组合结果由一个简单的对象构成，其中定义了一个 register()方法并以 name 作为键。

- register()：register()是一个回调方法，当插件在 app.js 中注册服务器而被绑定时将被显式地调用。该方法在服务器初始化阶段即被调用，而不是请求事件时。因此，我们移动了'then-able'回调中的 server.start()方法，以便服务器在加载插件时处于等待状态。简而言之，这里采用了内建的 Promise 库方法处理异步问题。所有修改之处均参考了 app.js 文件，如下所示。

```
const Hapi = require('hapi');
const constants = require('./constants');
const routes = require('./routes');
const plugins = require('./plugins');
const port = constants.port;
const server = new Hapi.Server({ port });

server.route(routes);
server.register(plugins.logRequest)
.then(() => {
  server.start();
})
.catch((err) => {
```

```
  console.log("error", err);
})

console.log(`Server running at: ${server.info.uri}`);
```

由于使用了 Promises 处理插件的异步问题，因而当抛出任何错误时，catch()方法均会对此进行处理。

- name：插件是用一个 name 属性定义的，以供 hapi 进行标识。每个自定义插件都是在使用之前命名的，它是一个强制属性。

当在 hapi 服务器中设置了插件后，通过重新启动服务器并通过浏览器请求即可运行代码。

查看终端并可得到下列输出结果。

```
A plugin got called!
Server running at: http://brunos-MacBook-Pro.local:3300
```

根据 hapi v17 所发布的文档，其中涵盖了下列属性并可在实现某个插件时加以使用。

```
{ register, name, version, multiple, dependencies, once, pkg }
```

下面讨论一些相对高级的话题，并编写一个插件以在每次接收一个请求时记录终端控制台中的 URL，具体步骤如下。

（1）使用 register()方法的服务器实例监听插件中的 onRequest 事件。Node.js 是使用事件驱动模型作为其主干来实现的。当某个请求在运行期内通过服务器所接收，将会触发 onRequest，如下所示。

```
server.ext('onRequest', (request, reply)=>{
  console.log("Listening to request!");
})
```

（2）当前，若重新启动服务器并访问 URL，将会得到一条错误消息，其原因在于，我们尚未指定事件循环下一步执行的任务。当处理请求并将其传递至对应的控制器方法时，需要使用包含 reply.continue 的 return 语句，如下所示。

```
server.ext('onRequest', (request, reply)=>{
  console.log("Listening to request!");
  return reply.continue;
})
```

continue 属性被实现为 reply 实例的属性，但是由数据组成的，按照新的 ECMAScript，这是一种符号数据类型。当在控制台中输出 reply 实例时，可看到一组属性，例如 close、abandon 等，进而可指定事件循环所执行的任务。

（3）至此，我们已经成功地实现了某种机制以处理所有请求。最后，我们需要从请求实例中提取 URL，对应代码如下所示。

```
//plugin.js
exports.logRequest = {
  register(server, options){
    console.log("A plugin got called!");
    server.ext('onRequest', (request, reply)=>{
      console.log("Listening to request!");
      const path = request.url.path;
      const target = request.url.query.target;
      console.log(`The target is ${target} for url: ${path}`);
      return reply.continue;
    })
  },
  name: "logRequest"
}
```

在将请求实例输出至控制台后，可以看到所提供的全部属性。当前，我们使用了 URL 属性提供所有的请求 URL 数据，例如可以通过 URL 传递的 URL 路径和查询参数。相应地，我们将使用 target 键作为/greetings URL 中的查询参数。该 URL 形如 http://localhost:3300/greetings?target=developers。

利用 npm start 重新启动服务器，可得到如图 8.3 所示的结果。

```
> node app.js

A plugin got called!
Server running at: http://brunos-MacBook-Pro.local:3300
Listening to request!
The target is undefined for url : /greetings
Listening to request!
The target is undefined for url : /favicon.ico
```

图　8.3

这类基于 JavaScript 对象的插件组合将会导致 App 的增量开发以及代码的模块化结果。对于较大的项目，这将有助于代码的可维护性。

## 8.5　使用 POSTMAN 测试 API

我们所创建的 API 端点（http://localhost:3300/greetings）是一个 GET 方法请求类型的 API 调用，并可方便地在浏览器 URL 定位器中直接测试 API 端点，其原因在于，当从浏

览器 URL 定位器中直接访问任何 URL 输入时，默认状态下将执行一个 GET 方法请求。那么，对于 POST、PUT、DELETE 来说，情况又当如何？当然，浏览器可以发出此类请求，但并不是在默认状态下，也不是通过 URL 定位器中的直接输入；而是通过 AJAX 请求或表单 POST 请求。

在这种情况下，如果创建了一个 API 服务器，并打算测试 URL 端点相对于其他请求方法的正确性时，我们将无法对此进行直接操作。此时，需要在客户端编写一些 JavaScript 代码进而生成此类 API 调用；或者使用诸如 POSTMAN 这一类 REST 客户端生成 API 调用。

### 8.5.1　使用 POSTMAN 测试 hapi 服务器调用

本节将针对 hapi 服务器 API 的用例安装 POSTMAN，并对其进行测试。对此，可执行以下步骤。

（1）访问 https://www.getpostman.com/。

（2）根据操作系统下载该 App 并安装。

图 8.4 显示了 POSTMAN App 的外观。

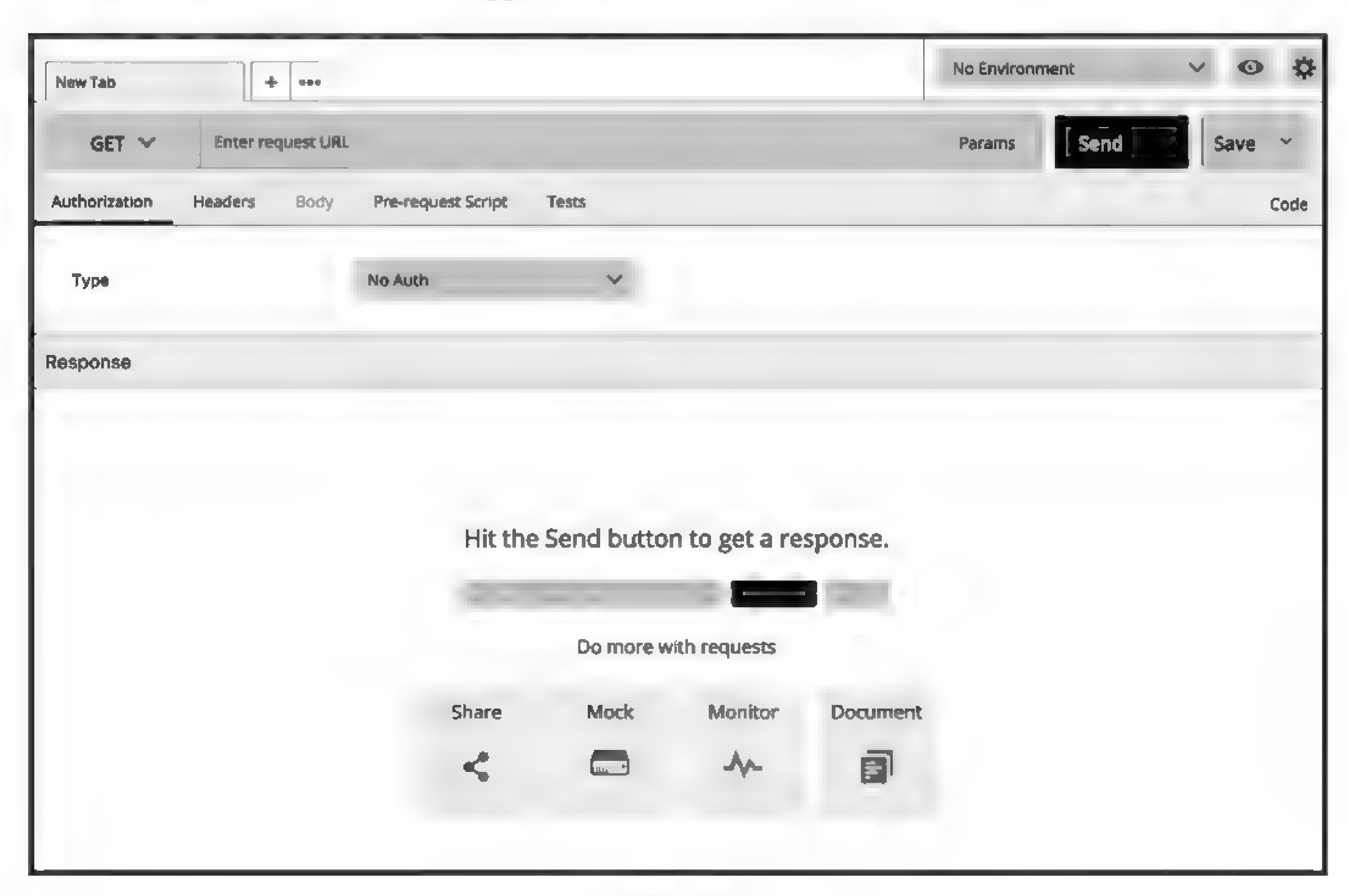



图　8.4

对于该客户端所生成的 HTTP 请求，现在只须输入所需的详细信息即可，具体步骤如下。

（1）输入请求数据的 URL。在当前示例中为 http://localhost:3300/greetings。

（2）上述请求 URL 输入表示为请求方法选择项下拉菜单。这里须检测期望的请求方法是否已被选取。默认状态下，将选取 GET 方法；此外，还可选择其他所需方法。

（3）检查是否需要发送 URL 或头参数。在当前示例中，URL 参数为 target。图 8.5 显示了全部所需数据。




图 8.5

单击 Send 按钮并接收请求数据。这便是我们在开发过程中测试和调试每个 API 的方法，该方法节省了大量的调试时间，并可在必要时按需执行异步请求。

## 8.5.2 POSTMAN 下的 JSON

POSTMAN App 的目的不仅是提供内容（虽然这是一个十分重要的功能），还可将所有 API 端点整合至一个集合中并一起运行。

除此之外，还可以使用浏览器的 POSTMAN 拦截器扩展来拦截 POSTMAN 客户端。这可帮助我们在 POSTMAN App 中直接查看浏览器请求，且无须任何手动交互操作。对于可移植性来说，甚至还可导出端点集合。

此类特性在任何 API 服务器开发过程中均十分有用。当访问 POSTMAN LABS 时（对应网址为 https://github.com/postmanlabs），可以看到，全部的 POSTMAN 存储库均为开源且供社区所用。此外，JSON 的广泛用途也可在此得到印证，特别是 POSTMAN 集合存储库，对应网址为 https://github.com/postmanlabs/postman-collection。

接下来将逐步考查集合特性，以便了解 App 下 JSON 的应用方式。

（1）App 右侧包含了两部分内容，即 History 和 Collection。如前所述，当生成 API 调用时，该调用将记录于 App 的 History 部分。下面导航至 History 部分并单击选项按钮（采用 3 个点表示），如图 8.6 所示。

从弹出菜单中选择 Save Request 选项，将会显示一个 SAVE REQUEST 对话框。

（2）此处可以保留与名称相同的请求 URL，或者将其更改为类似于 Greeting url。此外，我们还需要创建一个可以保存请求的集合，该操作可在同一个对话框中完成，如图 8.7 所示。

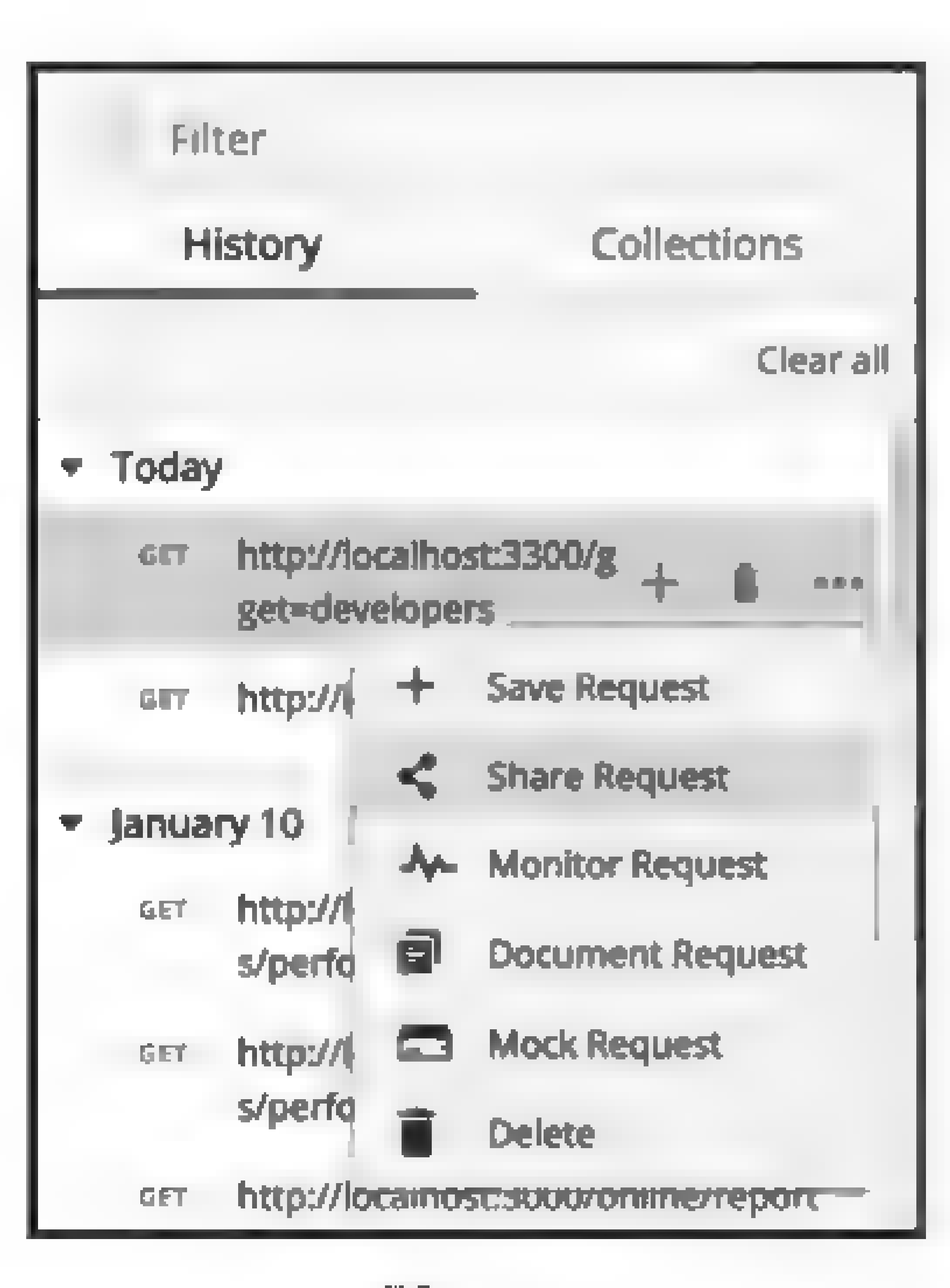



图　8.6




图　8.7

单击 Save 按钮，并将请求数据保存至所选的集合中。

（3）待集合保存完毕后，可访问该集合的选项并从弹出菜单中选取 Export，进而通过所需的选项导出数据。随后，JSON 文件将保存至当前机器上。当查看该文件时，其内容如下所示。

```
{

  "variables": [],
  "info":
  {
    "name": "node test app",
    " postman id": "5f507e82 d39d e96e edc5 efa2fd3c11b1",
    "description": "",
    "schema":
    "https://schema.getpostman.com/json/collection/v2.0.0/collection.json"
  },
  "item": [
    {
      "name": "Greetings url",
      "request":
      {
        "url":
        {
          "raw": "http://localhost:3300/greetings?target=developers",
          "protocol": "http",
          "host": [
            "localhost"
          ],
          "port": "3300",
          "path": [
            "greetings"
          ],
          "query": [
            {
              "key": "target",
              "value": "developers",
              "equals": true,
              "description": ""
            }
          ],
          "variable": []
        },
        "method": "GET",
        "header": [],
        "body":
        {},
        "description": ""
      },
```

```
      "response": []
    }
  ]
}
```

这里，所有数据都是纯 JSON 格式的。因此，可以说 POSTMAN 是以 JSON 的形式维护数据的。这种格式还提供了应用程序中的易读性和数据可移植性。

读者可访问 https://github.com/bron10/json-essentials-book/tree/master/chapter%209 以查看代码库。

## 8.6 本章小结

基于 JavaScript 的可配置服务器具有许多优点。虽然在开始阶段需要花费一些时间配置相关对象，但它提供了极大的灵活性、可复用性，以及代码的可维护性。相应地，hapi 服务器似乎从一开始就为可伸缩性做好了准备。而且，我们无须处理来自服务器或其内容的全部数据响应，仅须将数据返回回调处理程序即可，hapi 将亲自识别其类型。因此，为每种响应类型（例如 HTML、JSON 或任何字符串）定义内容类型，或者查找 npm 数据包所花费的时间将大大减少。

最后，本章还讨论了采用 POSTMAN 客户端的调试处理及其应用。创建一个完整的 hapi 服务器系统需要大量使用 JSON，这也是在服务器端应用 JSON 的重要步骤之一。第 9 章将讨论 NoSQL 数据库 MongoDB，并将其与当前 App 集成以实现持久化存储。

# 第 9 章　在 MongoDB 中存储 JSON 文档

NoSQL 数据库正在成为下一个重大事件。从地理空间数据领域到小型企业，NoSQL 数据库的实现呈指数级增长，这取决于 NoSQL 在管理数据时体现的灵活性，本章也将对此展开讨论。

NoSQL 数据库并不是一类关系型数据库。也就是说，数据库是非结构化的，而非表格形式。这一类 NoSQL 数据库可容纳各种数据模型，例如图、单级或多级键-值对、JSON 文档等。其中，MongoDB 即一类 NoSQL 数据库，并采用了 BSON 文档存储。本章主要涉及以下主题。

- ❑ 配置 MongoDB 并与 hapi App 集成。
- ❑ JSON 和 BSON。
- ❑ 插入 JSON 文档。
- ❑ 检索 JSON 文档。
- ❑ MongoDB 中基于 JSON 的模式。

## 9.1　配置 MongoDB

对于 Linux、macOS 和 Windows 7（以及更高版本）等操作系统来说，MongoDB 的安装过程较为简单。其中，Windows 7 版本需要安装一些补丁程序。我们不会关注每个操作系统上的每个安装步骤，因为这并不是本章所要讨论的重点内容。因此，我们将使用 Homebrew 在 macOS 上安装 MongoDB。

Windows 用户可访问 https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/以了解更多信息。

Linux 用户可访问 https://docs.mongodb.com/manual/tutorial/install-mongodb-on-linux/以了解更多信息。

macOS 用户可访问 https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/以了解更多信息。

Homebrew 是一个 macOS 中的包管理器。这里应确保首先安装 Homebrew（对应网址为 https://brew.sh/），以便使用 brew 命令。MongoDB 的配置过程具体如下。

（1）利用下列命令安装 mongodb 包。

```
brew install mongodb
```

（2）待安装完毕后，在运行 MongoDB 服务之前，需要向/data/db 目录授予一定的许可权限。这一类目录经配置后将保存 MongoDB 数据库。鉴于该数据目录创建为根目录，因而需要针对 MongoDB 应用程序用户提供相应的权限，以访问该目录。对此，可采用下列命令。

```
sudo chown -R `id -un` /data/db
```

这里采用递归方式设置了 data 目录中所有目录的访问权限。此处建议使用 chown 命令而不是 chmod 命令，chmod 命令也可以授予其他用户访问权限。

（3）在 Terminal 中运行 mongod，将会得到与 MongoDB 连接及其存储引擎 WiredTiger 相关的一些信息。

（4）待数据目录配置完毕后，还需要检测 MongoDB 在系统上是否运行良好。对此，可执行下列命令。

```
mongod
```

对应结果如图 9.1 所示。

图　9.1

其中，MongoDB 在默认端口 27017 等待输入请求。需要注意的是，为了提升 MongoDB 数据库系统上 App 的可扩展性，WiredTiger 是 MongoDB 所需的一种高效的存储引擎，并在 3.2 版本之后与 MongoDB 实现了集成。另外，图 9.1 中还显示了当前 MongoDB 的版本

号，即 db version v3.6.2。

在配置了 MongoDB 后，下一步是将 hapi App 与 MongoDB 进行连接，以实现持久化存储。

## 9.2　连接 hapi App 与 MongoDB

第 8 章曾讨论了 hapi 服务器，本节将讨论 hapi 与 MongoDB 之间的连接方式。

MongoDB 通过 npm 提供了客户端 MongoDB 节点模块。对此，需要在之前创建的 node-test-app 目录中安装 mongodb 客户端。相应地，我们可直接从 GitHub 中获取代码库，对应网址为 https://github.com/bron10/json-essentials-book。

随后，在 Terminal 中运行下列命令。

```
npm install mongodb --save
```

上述命令将在 hapi 服务器 App 的节点模块中安装 mongodb 客户端，并将其自身注册于 package.json 中，如下所示。

```
{
  "name": "test-node-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node app.js",
    "new-start": "node app.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "handlebars": "^4.0.11",
    "hapi": "^17.2.0",
    "mongodb": "^3.0.2"
  }
}
```

当在节点模块中设置了 mongodb 包后，即可在代码中对其加以使用。对此，我们需要使用 mongodb 以及连接凭证对其进行配置，如下所示。

```
const Hapi = require('hapi');
const constants = require('./constants');
```

```
const routes = require('./routes');
const plugins = require('./plugins');
const port = constants.port;
const server = new Hapi.Server({ port });
const MongoClient = require('mongodb').MongoClient;
MongoClient.connect(constants.mongodb.url, function(err, client) {
  if (err)
    throw err;
  console.log("Connected successfully to mongodb server");
});
server.route(routes);
server.register(plugins.logRequest)
  .then(() => {
    server.start();
  })
  .catch((err) => {
    console.log("error", err);
  })
console.log(`Server running at: ${server.info.uri}`);
```

下面对上述代码加以逐一解释。

（1）包含 require('mongodb')。MongoClient 提供了一个 mongo 实例，进而提供了一个名为 connect 的原型方法，负责构建与 MongoDB 客户端之间的连接。

（2）connect()方法接受 mongo 服务器 URL。注意，这里提供了一个基于 JSON 的解决方案，并实现为 constant.js 配置提供程序。我们通过地址 constants.mongodb.url 访问源自 JSON 的 URL 数据。该 URL 地址由 IP 地址构成（用作本地主机）且默认端口为 27017，如下所示。

```
//constants.js
module.exports = {
  "port": 3300,
  "audience": "readers",
  "mongodb": {
    "url": "mongodb://localhost:27017"
  }
}
```

（3）MongoClient.connect 的第二个参数表示为一个回调函数，并在 mongo 尝试构建与 MongoDB 之间建立连接时被调用。鉴于代码的同步特征，此处使用了一个回调。连接构建过程中的错误或成功结果将在该回调中予以处理。

当利用 npm start 启动服务器时，将得到下列输出结果。

```
Server running at: http://brunos-MacBook-Pro.local:3300
 Connected successfully to mongodb server
```

为了以更加高效的方式处理异步代码，我们将实现相应的 Promise，下面将对其进行简要的回顾。在第 3 章曾介绍了 Promaise，它表示为一个样板代码，与常规的回调方式相比，可更有效地处理异步数据。此外，Promise 还提供了较好的代码结构，并维护代码的可读性。下面将对 mongodb 客户端实现 Promise 化，如下所示。

```
MongoClient.connect(constants.mongodb.url)
  .then(function() {
    console.log("Connected successfully to mongodb server")
  })
  .catch(function(err) {
    console.log("An error occurred while connecting to mongodb!", err)
})
```

注意，此处在 App 初始化阶段即建立了与 mongodb 间的连接。这里不建议关闭 MongoDB 连接，因为这很容易导致内存泄漏问题，所以需要采取以下措施。

（1）将 mongodb 连接方法移至 routes.js 中，并在每次请求时进行连接。

（2）在每次请求时完成了 mongodb 连接后，即关闭该连接。

当在文档上执行相关操作时，我们将实现上述功能。接下来将介绍 MongoDB 中的集合和文档。

## 9.3　JSON 和 BSON

前述章节详细介绍了 JSON，本节将从以下 3 个方面探讨 BSON。

- MongoDB 驱动程序将输入的 JSON 转换为称为 BSON 的、二进制编码的 JSON，并将其传递至存储引擎（当前为 WiredTiger）中，如图 9.2 所示。

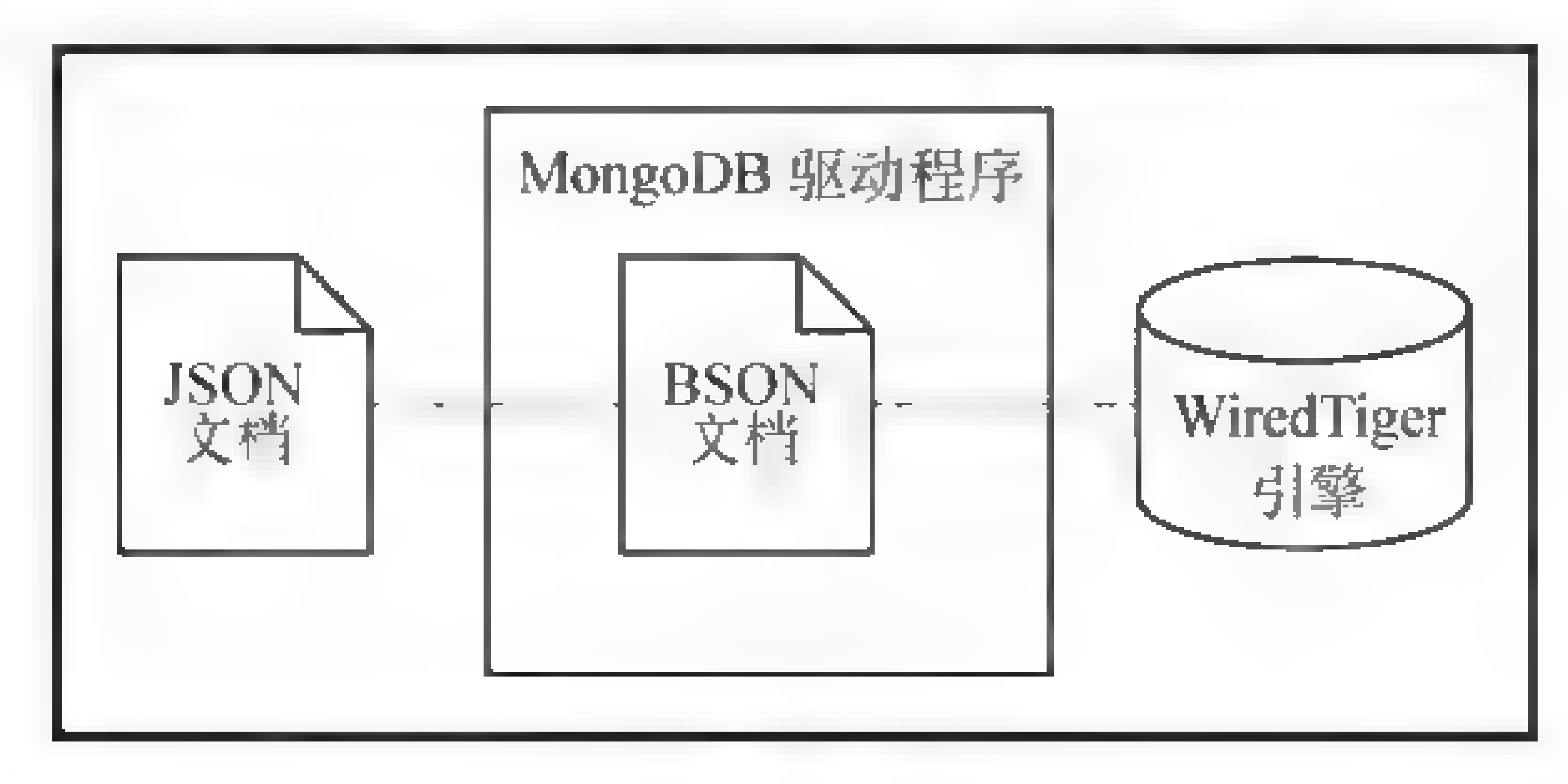



图　9.2

在图 9.2 中，输入的 JSON 被转换为称作 BOSN Document 的二进制序列化 JSON 实体，并存储于数据库中。

- BSON 规范涵盖了诸多优点。例如，与 JSON 相比，BOSN 占用较少的内存空间；BOSN 是可遍历的，同时还可对其进行查询；另外还可快速被解析以供其他所支持的语言使用。
- 当检索 BOSN 文档时，BOSN 规范中的数据类型通过 MongoDB 驱动程序转换为本地语言数据类型，以供开发人员使用。

开发人员无法直接参与 BSON 事物，且仅可与 JSON 协同工作。MongoDB 驱动程序负责处理一切事物。

关于 BSON，读者可访问 http://bsonspec.org/以了解更多内容

## 9.3.1 集合

在结构化查询语言中，记录或元组将被存储至表中；而 BSON 文档则存储于 MongoDB 的集合中。这里，集合表示为一个或多个 BSON 文档的分组，同时也是指向文档集群的引用。稍后我们将对文档的视图加以讨论。

## 9.3.2 MongoDB shell

MongoDB 提供了内建的 shell 或 Terminal，以访问数据库并执行原始操作。下面将把一个简单的 JSON 数据插入 customers 集合中。

假设 MongoDB 服务已经运行于当前系统上，此处打开一个新的 Terminal。随后，输入命令 mongo 打开一个新的 mongo shell，后面跟着另一个命令 show dbs。图 9.3 中显示了 Terminal 中的输出内容。

其中，show dbs 提供了之前配置的、MongoDB 的/data/db 目录中的数据库列表。

图 9.3 还显示了另一条命令 use test。use 是一条功能强大的命令，可创建新的数据库并将其选为所用；或者选取现有的数据库。在上述示例中，我们选择了一个已有的数据库。

在选取了数据库之后，接下来需要创建一个集合。在 MongoDB 中，我们无须维护任何模式，开发人员或者管理员负责根据需要构建一个集合，这种灵活性有效地减少了数据库的设计时间。

如前所述，我们将集合定义为文档的引用。因此，在将 JSON 文档（BSON）插入至集合之前，集合并无实际用处；或者在这种情况下集合尚不存在。下面将插入一个新的 JSON 文档，并创建一个名为 customer 的集合，如下所示。

```
db.customers.insert({
  "cust id": 1,
  "firstname": "John",
  "lastname": "Doe",
  "Address": {
    "pincode": "900001",
    "street": "3305 Tenmile",
    "city": "LA"
  }
})
```

上述代码将一个包含客户信息(例如名称和地址)的 JSON 文档插入 customer 集合中。在 shell 中作为命令运行上述代码片段。

```
Last login: Fri Feb  2 09:23:10 on ttys007
brunos-MacBook-Pro:~ bruno$ mongo
MongoDB shell version v3.6.2
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.2
Server has startup warnings:
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten]
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten] ** WARNING: Access control is not enabled for the database.
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten] **          Read and write access to data and configuration is unrestricted.
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten]
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten] ** WARNING: This server is bound to localhost.
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten] **          Remote systems will be unable to connect to this server.
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten] **          Start the server with --bind_ip <address> to specify which IP
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten] **          addresses it should serve responses from, or with --bind_ip_all to
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten] **          bind to all interfaces. If this behavior is desired, start the
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten] **          server with --bind_ip 127.0.0.1 to disable this warning.
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten]
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten]
2018-02-02T09:33:41.797+0530 I CONTROL  [initandlisten] ** WARNING: soft rlimits too low. Number of files is 256, should be at least 1000
> show dbs
admin        0.000GB
local        0.000GB
restaurants  0.000GB
test         0.000GB
> use test
switched to db test
>
```

图　9.3

随后，通过集合实例 db.customers 中的 find()方法可检索全部数据。数据的检索代码片段如下所示。

```
db.customers.find().pretty()
```

图 9.4 汇总了在集合上执行的整体操作。

在 mongo shell 中，db 表示为引用集合的引用变量。相应地，集合提供了某些公有方法，例如 insert()、find()、drop()等。这也体现了 JSON 数据基于 mongo shell 的管理方式。接下来将使用代码中的 mongo 客户端模块，并通过编程方式访问集合文档。

```
> db.customers.insert({"cust_id":1,"firstname":"John","lastname":"Doe","Address":{"pincode":"900001","street":"3305 Tenmile","city":"LA"}})
WriteResult({ "nInserted" : 1 })
> db.customers.insert({
...     "cust_id": 2,
...     "firstname": "smith",
...     "lastname": "Tiger",
...     "Address":
...     {
...         "pincode": "900002",
...         "street": "3305 Twentymile",
...         "city": "LA"
...     }
... })
WriteResult({ "nInserted" : 1 })
>
> db.customers.find().pretty()
{
        "_id" : ObjectId("5a73e93236b515b8decde684"),
        "cust_id" : 1,
        "firstname" : "John",
        "lastname" : "Doe",
        "Address" : {
                "pincode" : "900001",
                "street" : "3305 Tenmile",
                "city" : "LA"
        }
}
{
        "_id" : ObjectId("5a73eb2430b515b8decde685"),
        "cust_id" : 2,
        "firstname" : "smith",
        "lastname" : "Tiger",
        "Address" : {
                "pincode" : "900002",
                "street" : "3305 Twentymile",
                "city" : "LA"
        }
}
>
```

图　9.4

## 9.4　插入一个 JSON 文档

与 SQL 写操作相比，NoSQL 写入操作速度更快。这是因为我们不必从一开始就维护任何模式及其相关的数据类型。

下面继续讨论 test-node-app，并实现一个 POST API 调用，进而在集合中设置客户的数据。相关步骤如下。

（1）之前曾使用 app.js 文件中的 MongoDB 连接实例。此处首先移除 app.js 中的下列代码片段，并将代码保存在临时文件中，以供后续操作使用。

```
const MongoClient = require('mongodb').MongoClient;
MongoClient.connect(constants.mongodb.url)
  .then(function() {
    console.log("Connected successfully to mongodb server")
  })
  .catch(function(err) {
    console.log("An error occurred while connecting to mongodb!", err)
  })
```

（2）创建新的中间件，使用现有的 db 连接并针对特定集合进行查询。对此，在 plugin.js 文件中添加下列代码。

```
exports.mongoConnect = {
  register(server, options){
    return MongoClient.connect(constants.mongodb.url)
      .then(function(client){
        console.log("Connected successfully to mongodb server");
        server.ext('onRequest', (request, reply)=>{
          request.dbInstance = client.db('test');
          returnreply.continue;
        })
        return;
    })
    .catch(function(err){
      console.log("An error occurred while connecting to mongodb!", err)
    return
    })
  },
  name: "mongoConnect"
}
```

在成功建立连接后，可将数据库实例保存至请求事件中，以及 App 共享的请求单例实例中。

也就是说，将 db 实例保存至 request 中。这里，dbInstance 提供了一种方式，可在请求生命周期内传递和使用同一个实例。接下来，我们将在下一步创建的处理程序中使用该 db 实例。

（3）使用 db 实例并通过新的 API 处理程序查询集合测试，这可视为 http://localhost:3300/customer/add URL 上的 POST 调用。下列内容显示了可配置的对象代码，并于随后插入 routes.js 中。

```
{
  method: 'POST',
  path: '/customer/add',
  handler(request, h){
    const dbInstance = request.dbInstance;
    const requestBody = request.payload;
    return dbInstance.collection('customer')
    .insert(requestBody)
      .then((insertedStatus) =>{
        return "customer added successfully";
```

```
    })
    .catch((err) =>{
      return new Error(err);
    })
  }
}
```

当采用 customer 的集合实例时，我们使用了 insert()方法存储负载中的请求数据。

（4）准备 POSTMAN 或其他 REST 客户端中的请求数据。请求数据的相关格式如下所示。

```
{
  "firstname": "Bron",
  "lastname": "Dave",
  "Address": {
    "pincode": 123456,
    "street": "Thirtymile",
    "city": "LC"
  }
}
```

在 REST 客户端 App 或者 POSTMAN 浏览器扩展中，访问 API 端点，响应结果如图 9.5 所示。




图　9.5

确保接收到的响应状态代码为 200，其中包含了成功消息，同时表示操作已经完成。

## 9.5　检索 JSON 文档

通过所选集合上的 find()方法，可执行相应的检索操作。相应地，此处将继续添加客户功能，并在成功响应时提供已添加客户的列表来执行此项操作。对此，考查下列代码。

```
{
  method: 'POST',
  path: '/customer/add',
  handler(request, h){
    const dbInstance = request.dbInstance;
    const requestBody = request.payload;
    const customerCollection = dbInstance.collection('customer');
    return customerCollection.insert(requestBody)
      .then((insertedStatus) =>{
        return customerCollection.find(
          {})
          .toArray()
          .then((customerList) =>{
            return {
              message: "customer added successfully",
              customerList
            }
          })
     })
     .catch((err) =>{
       return new Error(err);
     })
  }
}
```

针对上述代码片段，需要注意以下几点内容。

- 当采用空对象作为参数时，find()方法将返回所有已出现的选择结果。在当前示例中为客户信息。
- 使用 cursor.toArray()方法将结果数据转换至一个数组中。

ⓘ 读者可访问 https://docs.mongodb.com/manual/reference/method/js-cursor/以查看更多游标（cursor）方法。

这里，游标是指向结果或引用的指针。find()方法返回对结果的引用；在执行游标方

法时，可从中析取相关数据。

- ❑ 最后返回一个有效对象，其中包含了成功消息和 customerList 数据。

最终结果如图 9.6 所示。

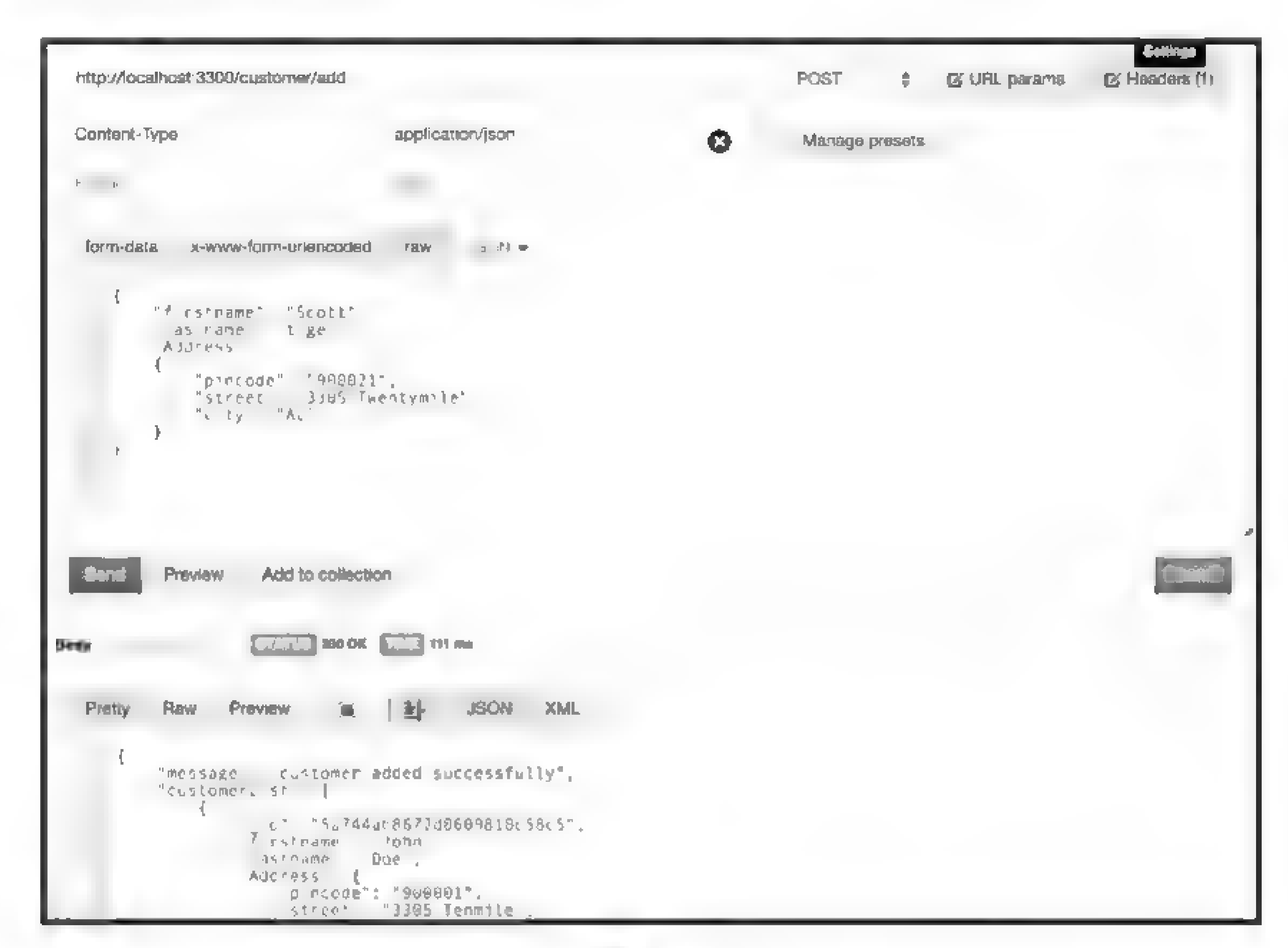



图　9.6

## 9.6　MongoDB 中基于 JSON 的模式

在应用程序开发之前，可能需要使用数据库的模式。或者，如果我们希望针对全部数据集设置数据类型，情况又当如何？

此时，我们打算构建和验证 NoSQL 数据。针对于此，我们需要寻找一种框架，它提供了一个围绕 mongodb 的包装器。

mongoose 是一个领先的第三方框架，它提供了所有基于模式的验证、虚拟属性等。

接下来将讨论如何利用 mongoose 准备和验证基于 JSON 的模式，相关步骤如下。

（1）通过 npm 命令安装 mongoose，如下所示。

```
npm install mongoose --save
```

（2）考虑到将在应用程序中实现 mongoose，因而可采用 mongoose 提供的包装器方法连接 MongoDB。

简单地注释掉 plugins.js 中的 mongodb 客户端并包含以下内容。

```
const MongoClient = require('mongoose');
```

这将在 MongoClient 常量中分配 mongoose 模块方法。除此之外，另一个细微的变化则是无须设置请求上的 dbInstance，mongoose 为我们完成了全部工作。接下来需要注释掉 server.ext 事件监听器，如下所示。

```
/**
server.ext('onRequest', (request, reply)=>{
  request.dbInstance = client.db('test');
  return reply.continue;
})**/
```

（3）在 mongoose 中，模式提供了 mongodb 模型的蓝图。它们作为一种结构，以一致性的方式保存传入的集合数据。下面将在一个新文件 model.js 中为客户数据库创建一个简单的模式，如下所示。

```
const mongoose = require('mongoose');
module.exports.customers = mongoose.model('customers', {
  "firstname": String,
  "lastname": String,
  "Address": { type: mongoose.Schema.ObjectId, ref: 'customers_address'
  }
});
const Address = mongoose.model('customers_address', {
  "pincode": Number,
  "street": String,
  "city": String
});
```

上述代码针对 customer 集合创建了一个 customers 模型。就 RDBMS 来说，我们针对客户表创建了一个客户模式。随后可利用数据类型属性集合（例如类型、引用、默认值等）填充键，例如地址。或者，也可简单地定义数据类型，例如上述代码片段中的 firstname 键。

需要注意的是，这里设置了一个 Address 引用类型。通过使用数组或引用数据类型，mongoose 还支持模式的嵌套。在上述示例中，我们采用了类型引用。

接下来，分配给每个键的数据类型仅是定义它的方法，而不是执行实际验证的方法。因此，如果作为一个字符串发送 pincode，它将会被保存；但如果作为布尔值发送定义完毕的 pincode，并在有效载荷中为 pincode 请求一个字符串，这将抛出一个类型转换错误。当验证 pincode 时，可向 customer_address 模型添加一个简单的验证器，如下所示。

```
module.exports.customers address = mongoose.model('customers address', {
"pincode": {
  type: Number,
  validate: {
    validator: function(v) {
      return `${v}`.length === 6;
    },
    message: '{VALUE} is not a valid pincode!'
    }
  },
  "street": String,
  "city": String
});
```

一种较常使用的验证类型是自定义验证器方法。其间，只需传递一个函数，该函数返回传入的 pincode 编号的真值，验证工作随即完成。在上述示例中，我们检测了输入的 pincode 值（v）的长度是否为 6，否则将抛出一个错误，以表示未满足当前条件。这里，甚至可以使用 message 键定制错误消息。

（4）最后，利用 module.exports 导出客户的集合模式，以供后续操作使用。

（5）一切顺利后，即可构造控制器，以便处理输入和输出数据。此类操作将在 routes.js 中完成，如下所示。

```
{
  method: 'POST',
  path:'/customer/add',
  handler(request, h) {
     const requestBody = request.payload;
     return (async ()=>{
       try{
         /**
          * Native mongodb code
          * constdbInstance = request.dbInstance;
          * constcustomerCollection = dbInstance.collection('customer');
          * return customerCollection.insert(requestBody)
          **/
         const customersModel = models.customers;
         const customersModelInstance = new models.customers(requestBody);
         let error = new
           models.customers address(requestBody.Address).validateSync();
         if(error)
           return Boom.boomify(error, { statusCode: 422 });
         await customersModelInstance.save();
```

```
        const customerList = await customersModel.find({});
        return {
          message: "customer added successfully",
          customerList
        };
      }catch(e){
        throw Boom.boomify(e, { statusCode: 500 });
      }
    })();
  }
}
```

其中，JavaScript 对象表示为路由的配置信息，并由请求方法、请求 URL，以及构造保存数据逻辑的方法（称作处理程序）构成。

下面对上述代码逐一分析。

- Models.customers 提供了客户模式实例，它是由所有可以在请求生命周期中使用的 mongoose 方法构建的原型。在当前示例中，我们采用了 find()方法。
- 当同一 Models.customer 采用新的关键字和负载作为参数实例化时，所接收的实例将提供有效的负载，以及负载操作方法，例如 save()、update()或 delete()。
- 采用 validateSync()方法对 pincode 进行验证。在当前示例中，该操作呈异步状态。对于异步代码，可简单地使用基于回调的 validate()方法。
- 此处利用工具库 Boom 处理错误响应，这也是 hapi 强烈推荐的一种方式。

关于 Boom 的更多信息，读者可访问 https://github.com/hapijs/boom。

- 考虑到代码本质上呈异步状态，因而可通过 async-await 对其进行处理。正如 hapi v17 所述，这将是一种完全的异步/等待端到端行为。这里，异步/等待是框架间处理异步行为的一种最新方式，并可生成更加简洁的代码。
- 最后，向 hapi 处理程序返回了一个自调用函数。

接下来开始运行 App。

针对一些变化内容，需要重新启动服务器。对此，可通过 Ctrl+C 快捷键终止 Terminal 中处于运行状态的节点服务，并使用 npm start 对其再次启动。

除此之外，还需要准备浏览器客户端 POSTMAN 中的数据，并添加集合中新的客户数据，如图 9.7 所示。




图　9.7

关于 mongoose，读者可访问 http://mongoosejs.com/docs/guide.html 以了解更多信息。

## 9.7　本 章 小 结

我们正在成为一名 JavaScript 全栈开发人员，并领略了不同 Web 开发领域中的 JSON 实现。本章主要介绍了 NoSQL 数据库脚本机制，其间讨论了 MongoDB 的配置、BSON、MongoDB 集合上的基本操作，以及对象-模型框架 Mongoose。

第 10 章将继续讨论 JSON 的实现，即针对应用程序开发自动化脚本或任务管理器。

# 第 10 章　利用 JSON 配置任务管理器

当今，单元测试、代码集成和创建可部署的优化版本等阶段均是应用程序开发生命周期的主要部分。考查 GitHub 上的代码存储库：在使用任何库或模块之前，社区会查找测试用例（特别是 spec.js 文件）和构建状态百分比（是通过还是失败）。很明显，如果缺少这样的校验和任务，开发人员将无法为应用程序模块的稳定性和性能提供基准。本章主要涉及以下主题。

- ❑ 任务管理器的含义及其使用原因。
- ❑ gulp.js 的含义及其功能。
- ❑ 利用 Mocha 框架和 gulp.js 执行单元测试。
- ❑ 利用 JSON 配置自动化全部单元测试用例。

下面将逐一加以讨论。

## 10.1　任务管理器的含义

在 Web 脚本上下文环境中，任务管理器是一类库，以帮助我们方便地执行某些功能，并最小化工作量，具体如下。

- ❑ 缩减客户端应用程序 JavaScript 文件以提高总体性能。
- ❑ 在构建时配置应用程序模块选择方案。
- ❑ 在一条命令上构建一个版本，且仅包含单一 index.html 文件、CSS、资源数据以及简化的 JavaScript。
- ❑ 测试 BDD（行为驱动数据（Behavior Driven Data，BDD））断言。

本章将利用 Nide 的默认断言库分别测试 API，并通过 gulp 任务管理器库对其实现自动化。在第 9 章中，曾在 hapi 应用程序中创建了两个 API，接下来将对其进行测试。在创建任务之前，下面首先讨论 gulp.js 及其用途和功能。

## 10.2　gulp.js 简介

基本上讲，gulp.js 是一个 JavaScript 任务管理器。前面列出的所有任务都可使用 gulp.js 执行，包括打包文件、在编码后保存文件时自动刷新浏览器、运行单元测试文件，以及利用 Amazon API 进行部署。gulp.js 可执行很多重复性和枯燥的任务。

“gulp 是一个工具箱，用于在开发工作流中自动执行某些复杂、耗时的任务，从而可节省开发人员的大量时间”。

——gulpjs.com

下面考查特征列表，并揭示 gulp.js 为何如此受到大家的欢迎（与开源市场上的其他库相比，例如 grunt.js）。

- ❑ 针对任务实现的小型插件结构。
- ❑ 易于阅读、理解任务。
- ❑ 可集成任务并执行自动化操作。
- ❑ 全部 IO 操作均通过 node.js 流执行，因此，全部更改内容首先在内存中完成，随后一次性写入端点，进而使速度和性能得到较大的提升。
- ❑ 较好的通信支持和插件生态环境。

接下来考查 gulp.js 的安装过程，随后讨论 gulp.js 的编码机制。gulp.js 的安装步骤如下。

（1）在项目的根目录中创建名为 gulpfile.js 的 gulp 配置文件，这将有助于在根目录中管理项目文件。在当前示例中，hapi App（node-tes-app 目录）和 gulp 配置文件将位于同一级别，因此第 10 章曾创建了此类文件夹，读者可访问 GitHub 存储库予以查看，对应网址为 https://github.com/bron10/json-essentials-book。

（2）利用下列命令安装全部所需的数据包。

```
$ npm install gulp-cli -g
$ npm install gulp --save-dev
```

其中，gulp-cli 数据包可在命令行界面中使用 gulp。待安装过程完毕后，gulpfile.js 将处于就绪状态，下面开始于其中编写代码。

## 10.3　在 gulp.js 中创建任务

本节将创建第一个任务，即用户欢迎画面，相关步骤如下。

（1）首先在 gulpfile.js 中编写插件代码，如下所示。

```
const gulp = require('gulp');
gulp.task('default', () => {
  console.log("Greetings to Readers!");
})
```

在上述代码片段中，gulp 模块的 task()方法需要使用到两个参数。其中，任务的第一个参数为字符串，第二个参数表示为一个回调，并可于其中编写特定任务背后的实际逻辑

内容。

（2）返回至 Terminal 并运行 gulp 命令，对应输出结果如图 10.1 所示。

```
Last login: Mon Feb 19 12:45:11 on ttys009
brunos-MacBook-Pro:chapter 10 bruno$ gulp
[17:13:03] Using gulpfile ~/Documents/projects/js/book/json essential/codes/chapter 10/gulpfile.js
[17:13:03] Starting 'default'...
Greeting Readers!
[17:13:03] Finished 'default' after 155 μs
brunos-MacBook-Pro:chapter 10 bruno$
```

图　10.1

接下来添加另一项任务，并将其用作上一任务的依赖项，如下所示。

```
const gulp = require('gulp');

gulp.task('default', ['dependent-task'], () => {
  console.log("Greetings to Readers!");
})

gulp.task('dependent-task', () => {
  console.log("Greetings to all!");
})
```

（3）当利用 gulp 执行上述代码时，对应输出结果如图 10.2 所示。

```
brunos-MacBook-Pro:chapter 10 bruno$ gulp
[17:26:13] Using gulpfile ~/Documents/projects/js/book/json essential/codes/chapter 10/gulpfile.js
[17:26:13] Starting 'dependent-task'...
Greetings to all!
[17:26:13] Finished 'dependent-task' after 184 μs
[17:26:13] Starting 'default'...
Greetings to Readers!
[17:26:13] Finished 'default' after 108 μs
brunos-MacBook-Pro:chapter 10 bruno$
```

图　10.2

在图 10.2 所示的输出结果中，dependent-task 在 default 任务之前运行，这可通过图 10.2 得到验证，即查看 console.log()方法所记录的语句顺序。这也证实了可以采用依赖任务的策略按顺序执行任务。

接下来将讨论以下内容。

- 针对某个 API 编写一个单元测试。
- 利用 gulp 运行 API 单元测试。

hapi App 提供了 API 端点（第 9 章曾对此有所介绍）。这里，应确保 MongoDB 和 Node

app.js 均处于运行状态。接下来编写一个简单的单元测试，以生成调用并对数据进行检测。

单元测试用例将对具有特定功能的代码单元进行测试。在当前示例中，将测试一个简单的路由或 hapi App。这里，单元测试用例任务将被划分为多个测量代码块，以便可清晰地编写单元测试用例。

（1）创建 routes.spec.js 文件。此处，命名规则指定了将要测试的文件 routes.js。开源社区建议，应将测试文件以 spec 作为后缀。

（2）根据 routing.js 中出现的所有路由来划分单元测试用例，并将断言封装在某些测试驱动的代码中。此处，npm 模块 mocha.js 可帮助我们完成这一项任务。顺便说一下，编写一组代码来完成，并通过测试用例被称为测试驱动开发（Test Driven Development，TDD）。

（3）利用下列命令安装 mocha。

```
npmi mocha --save-dev
```

（4）当指定单元用例时，需要包含 spec 文件中的代码，如下所示。

```
describe('Test routes', () => {
  it('Testing GET: greetings', () => {
    console.log("Testing the /greeting route");
  })
})
```

（5）运行 mocha 测试命令，如下所示。

```
$ ./node_modules/mocha/bin/mocha routes.specs.js
```

（6）对应输出结果如图 10.3 所示。

```
brunos-MacBook-Pro:chapter 10 bruno$ ./node_modules/mocha/bin/mocha routes.specs.js


  Test routes
Testing the /greeting route
    ✓ Testing GET: greetings


  1 passing (7ms)
```

图　10.3

此处，我们准备了相应的代码块，以便能够使用相关的测试场景。另外，运行单元测试规范文件的命令看起来过于冗长，并可对其稍作修改，即在根级别添加/调整 package.json 文件中的 script 键，如下所示。

```
"scripts": {
  "test": "./node modules/mocha/bin/mocha routes.specs.js"
}
```

（1）在 Terminal 中运行 npm test 命令。

关于 Mocha 的更多内容，读者可访问 https://mochajs.org/#getting-started

接下来将使用一个更好的库，可以在执行 test 文件时请求当前 API。对此，可采用 npm 库中的请求节点模块。

（2）运行下列命令。

```
npm install request --save-dev
```

（3）包含 routes.spec.js 中的请求模块，如下所示。

```
const request = require('request');
```

（4）请求模块实例可满足完成异步 HTTP 请求所需的各项要求。目前，我们的最小需求是请求 URL、一个请求方法和处理响应的回调，如下所示。

```
const request = require('request');

describe('Test routes', () => {
  it('Testing GET: greetings', () => {
    console.log("Testing the /greetings route");
    request.get('http://localhost:3300/greetings', (err,
      httpResponse,body) => {
      if (err) {
         throw err;
      }
      console.log("statusCode", httpResponse.statusCode);
      console.log("body", body);
      done();
    })
  })
})
```

（5）运行 npm test 命令，对应输出结果如图 10.4 所示。

需要注意的是，当前我们接收到的 statusCode 为 200，body 为 hello readers。done 参数表示为一个函数，用于运行异步代码的 mocha。当异步代码执行完毕后，需要调用 done 函数，即调用异步请求回调中的 done 函数。

接下来将介绍 node.js API 提供的断言库。这里，断言提供了一种方式可区分期望数

```
brunos-MacBook-Pro:chapter 10 bruno$ npm test

> gulp-file@1.0.0 test /Users/bruno/Documents/projects/js/book/json essential/codes/chapter 10
> ./node_modules/mocha/bin/mocha routes.specs.js



  Test routes
Testing the /greetings route
    ✓ Testing GET: greetings


  1 passing (18ms)

statusCode 200
body hello readers
```

图　10.4

据和实际数据。当在开发过程中使用断言时，可针对应用程序运用 BDD 测试方法，从而缩减开发阶段中 bug 的范围。下列代码显示了一些断言。

```
const request = require('request');
const assert = require('assert');

describe('Test routes', () => {
  it('Testing GET: greetings', (done) => {
    console.log("Testing the /greetings route");
    request.get('http://localhost:3300/greetings', (err,
      httpResponse,body) => {
      if (err) {
          throw err;
      }
      assert.equal(httpResponse.statusCode, 200);
      assert.ok(body == 'hello readers');
      done();
    })
  })
})
```

上述代码包含了 Node API 提供的 assert 模块。其中，assert.equal()方法用于检测两个操作数的相等性。在当前示例中，将以此检测响应 statusCode（利用 HTTP 响应的期望状态码所接收）。除此之外，它还将作为消息接收第三个参数，并在断言失败时予以显示。assert.ok()方法则接收一个真值作为参数，以完成有效的断言。在我们的示例中，如果响应体无法以 hello reader 作为响应，则断言失败。

在将期望的 statusCode 输入修改为 400 后（原值为 200），即可对断言失败进行测试，如图 10.5 所示。

```
brunos-MacBook-Pro:chapter 10 bruno$ npm test

> gulp-file@1.0.0 test /Users/bruno/Documents/projects/js/book/json essential/codes/chapter 10
> ./node_modules/mocha/bin/mocha routes.specs.js


  Test routes
Testing the /greetings route
    1) Testing GET: greetings


  0 passing (29ms)
  1 failing

  1) Test routes
       Testing GET: greetings:

      Uncaught AssertionError [ERR_ASSERTION]: 200 == 400
      + expected - actual

      -200
      +400

      at Request.request.get [as _callback] (routes.specs.js:11:11)
      at Request.self.callback (node_modules/request/request.js:186:22)
      at Request.<anonymous> (node_modules/request/request.js:1163:10)
      at IncomingMessage.<anonymous> (node_modules/request/request.js:1085:12)
      at endReadableNT (_stream_readable.js:1056:12)
      at _combinedTickCallback (internal/process/next_tick.js:138:11)
      at process._tickCallback (internal/process/next_tick.js:180:9)
```

图　10.5

当前测试用例已准备就绪。类似地，还可针对/customer/add API POST 调用编写一个API 单元测试，如下所示。

```
it('Testing GET: customer/add', (done) => {
  console.log("Testing the /customer/add route");
  let payload = {
    "firstname": "firstTest",
    "lastname": "lastTest",
    "Address": {
      "pincode": 111111,
      "street": "testmile",
      "city": "TC"
    }
  };

  request.post({
    url: `http://localhost:3300/customer/add`,
    json: payload
  }, (err, httpResponse, body) => {
    if (err) {
```

```
      throw err;
    }
    const filteredCustomerList =
    body.customerList.filter(function(customerData) {
      return (customerData.firstname == payload.firstname&&
      customerData.lastname == payload.lastname)
    })

    assert.equal(httpResponse.statusCode, 200);
    //The data inserted above should have atleast one customer instance
    assert.ok(filteredCustomerList.length> 1);
    done();
    })
  })

  assert.equal(httpResponse.statusCode, 200);
  //The data inserted above should have atleast one customer instance
  assert.ok(filteredCustomerList.length> 1);
  done();
  })
})
```

在上述代码片段中，我们已经了解了第一个断言。第二个断言表明，至少应出现一个实例（使用 API 端点插入客户列表集合中）。在这一场合下，该实例将匹配于客户实例（通过 firstname 和 lastname 获取）。

运行上述代码，将得到如图 10.6 所示的结果，表明两个测试均已通过。

```
brunos-MacBook-Pro:chapter 10 bruno$ npm test

> gulp-file@1.0.0 test /Users/bruno/Documents/projects/js/book/json essential/codes/chapter 10
> ./node_modules/mocha/bin/mocha routes.specs.js


  Test routes
Testing the /greetings route
    ✓ Testing GET: greetings
Testing the /customer/add route
    ✓ Testing GET: customer/add


  2 passing (35ms)
```

图　10.6

当前，单元测试已准备就绪，接下来将作为单项 gulp 任务对其进行集成，以对路由进行测试。gulp 中涵盖了一个较好的插件库，对于集成操作，我们将使用 gulp-mocha，并可通过下列命令安装。

```
npm install gulp-mocha --save-dev
```

此处将重点讨论 gulpfile.js 并创建一项新任务，如下所示。

```
gulp.task('test-routes', function() {
  return gulp.src('./routes.specs.js')
    .pipe(mochaPlugin({ reporter: 'spec' }))
});
```

当使用 mochaPlugin 时，需要将其包含在 gulpfile.js 的开始处，如下所示。

```
const mochaPlugin = require('gulp-mocha');
```

需要注意的是，此处使用了 pipe()方法，该方法可高效地将 spec 文件的源数据流传递至 mochaPlugin 中，以便执行当前测试。

当运行 gulp test-routes 命令时，单元测试通过 gulp 予以执行。接下来将通过依赖项策略创建一些自动化测试。

## 10.4　自动化测试

通过将 gulp 插件和 gulp 任务的功能集成在一起，可以创建一个连续的任务流，这些任务流可以根据具体需求并行或顺序地运行。在前述 gulp 任务依赖项策略的基础上，我们将采用 npm run-sequence 模块以顺序方式运行相关任务。当任务数量随时间增加时，这将有助于结构化对应任务，并降低问题的复杂度。

运行下列命令安装 run-sequence。

```
npm install run-sequence --save-dev
```

就应用方面来说，需要注意的是，每项 gulp 任务均应返回一个流或 Promise；或者利用回调加以处理。下面通过在流中创建 3 项简单的任务来证明这一点，并利用 gulp 自动化的强大功能。

（1）启用 hapi APP。

（2）测试 haip App 的 API。当进行测试时，确保 test-node-app 目录为 gulpfile.js 的根目录。另外，读者还可访问 https://github.com/bron10/json-essentials-book 以了解更多内容。

（3）终止 hapi App。

此类任务的对应代码如下所示。

```
const gulp = require('gulp');
const mochaPlugin = require('gulp-mocha');
const runSequence = require('run-sequence');
```

```
var exec = require('child process').exec;
let processInstance = undefined;

gulp.task('default', ['dependent-task'], () => {
  console.log("Greetings to Readers!");
})

gulp.task('dependent-task', () => {
  console.log("Greetings to all!");
})

//Start our hapi app
gulp.task('start-app', (cb) => {
  //Update the package.json for start script
  processInstance = exec(`npm start`);
  cb();
})

//Test the API of the hapi app
gulp.task('test-routes', () => {
  return gulp.src('./routes.specs.js').pipe(mochaPlugin({ reporter:
    'spec' }))
});

//Stop the hapi app
gulp.task('stop-app', (cb) => {
  processInstance.kill(0);
  process.exit(0);
  cb();
})
gulp.task('Test-API-Flow', function() {
  return runSequence('start-app', 'test-routes', 'stop-app');
});
```

在上述 gulp 任务中，我们利用 gulp 回调参数 cb 完成当前任务，或者返回一个流。这对于 runSequence 函数来说不可或缺，以便了解每项任务在下一项任务开始之前均已完成。下面逐一讨论每项新任务。

- start-app：该项任务通过运行 npm start 命令启用 hapi 服务器实例。由于 gulpfile.js 处于在终端中执行命令的根级别，因而该命令位于 gulpfile.js 的 package.json 中，而非 test-node-app 目录。另外，该命令还使用了 node.js API 提供的 child process 模块的 exec()方法，并可为生成 gulp 父进程的最新子进程。

- test-routes：表示为测试 hapi API 路由的 gulp 任务。
- stop-app：关闭两个进程（应用程序进程和 gulp 进程）的 gulp 任务。
- Test-API-Flow：该 gulp 任务只是对前面的所有任务进行排序。

在执行之前，应确保未运行服务器实例，并确保在启用 gulp 进程之前退出其他节点进程（尤其是 hapi 服务器进程）；否则将会产生一个地址使用错误。

运行 gulp Test-API-Flow 并查看 Terminal 中的自动化结果，如图 10.7 所示。

```
brunos-MacBook-Pro:chapter 10 bruno$ gulp Test-API-Flow
[12:19:18] Using gulpfile ~/Documents/projects/js/book/json essential/codes/chapter 10/gulpfile.js
[12:19:18] Starting 'Test-API-Flow'...
[12:19:18] Starting 'start-app'...
[12:19:18] Finished 'start-app' after 4.89 ms
[12:19:18] Starting 'test-routes'...
[12:19:18] Finished 'Test-API-Flow' after 17 ms

  Test routes
Testing the /greetings route
    ✓ Testing GET: greetings
Testing the /customer/add route
    ✓ Testing GET: customer/add

  2 passing (38ms)

[12:19:19] Finished 'test-routes' after 477 ms
[12:19:19] Starting 'stop-app'...
brunos-MacBook-Pro:chapter 10 bruno$
```

图　10.7

## 10.5　gulp JSON 配置

最后，本节将利用 JSON 配置 gulp 任务。下面创建一个名为 gulpconfig.json 的配置文件，该文件将对全部设置、源以及任务命名予以控制。尽管当前仅有一个路由 API 任务需要测试，但最好在高级阶段启动它。当应用程序 API 增加时，将很难在应用程序结构级别进行修改。gulpconfig 文件内容如下所示。

```
{
  "apiflow":
  {
    "name": "Test-API-Flow",
    "sequence": ["start-app", "test-routes", "stop-app"]
  },
  "routes":
  {
    "name": "test-routes",
```

```
    "src": "./routes.specs.js"
  }
}
```

修改 gulpfile.js 最终的下列任务。

```
//Test the API of the hapi app
gulp.task(gulpTasksConfig.routes.name, () => {
  return gulp.src(gulpTasksConfig.routes.src).pipe(mochaPlugin({
  reporter: 'spec' }))
});

gulp.task(gulpTasksConfig.apiflow.name, function() {
  return runSequence(...gulpTasksConfig.apiflow.sequence);
});
```

上述代码使用了配置文件的 gulpTaskConfig 实例。对此，应确保在 gulpfile.js 的开始处包含了下列内容。

```
const gulpTasksConfig = require('./gulpconfig.json');
```

在上面的代码中，我们学习了一种新的技术，将数组值作为参数隔离于 runSequence 函数调用之上。对应的操作符称作展开操作符，它由 3 个连续的点构成并作为配置键的前缀。展开操作符被称作一个序列并提供了一个数组。

## 10.6 本章小结

对于 JSON 的研究将我们推向了不同的学习阶段。本章讨论了单元测试和自动化。其间，我们使用了 gulp.js 这一重要的任务管理器工具，同时还考查了两种较为重要的单元测试概念，即基于 node.js 断言库的 BDD 测试，以及 JavaScript App TDD 框架 mocha.js。

这里，我们也强烈建议尽可能多地进行单元测试，这在很大程度上减少了错误出现的概率，并随着时间的推移提供了较好的代码可维护性。

*“如果您不喜欢对产品进行单元测试，那么，您的客户很可能也会放弃这类测试”*

*——佚名*

第 11 章将讨论实时系统和分布式系统中的 JSON 实现。

# 第 11 章　实时系统和分布式系统中的 JSON

截止到目前，我们使用 RESTful HTTP API 端点作为客户端和服务器之间的数据通信源。事实证明，HTTP 请求是获得可靠数据可用性的最佳方式。如果存在网络延迟问题，唯一的障碍便是响应时间。如果我们并不希望等待服务器响应，或者我们需要实时接收数据，情况又当如何？考虑以下场景：使用产品机器人进行一些简单的消息传递活动，或者为交付给工作人员的演示文稿进行屏播。在这种情况下，成功的唯一标准是数据的即时可用性和正确性。WebSocket 是一种为此类场景提供实时解决方案的 Web 技术协议。

另一个较为重要的事件则是分布式系统。一旦部署并运行了 Web 应用程序，就需要扩展网络资源以实现数据一致性，并提高远程节点之间的通信质量。在这种情况下，分布式系统解决方案用于网络间的数据管理。本章将学习 Apache Kafka，它提供了一个可伸缩的流处理系统。

本章主要涉及以下主题。

- ❑ 基于 Socket.IO 和 JSON 的实时通信。
- ❑ 配置 Socket.IO 服务器及其客户端。
- ❑ 基于 Apache Kafka 和 JSON 的分布式系统简介。
- ❑ 安装 Apache 并实现实时应用程序中的分布式系统概念。

## 11.1　基于 Socket.IO 的 JSON

Socket.IO 服务器的配置过程较为简单。本节将实现一个实时服务器，并提供连续的 HTTP 握手，同时通过 Socket.IO 框架监听请求。

本节将通过一个 pinboard 应用程序描述实时 JSON 的实现过程。任何连接到实时服务器的匿名用户都可以加入会话，并可查看 pinboard，同时添加他们喜欢的内容。

该应用程序涵盖以下两个阶段。

（1）设计 pinboard。

（2）通过 Socket.IO 库实现实时功能。

### 11.1.1　设计 pinboard

本节将设计一个 Web 界面，据此，用户可添加或查看插接板上的接口。回忆一下，

第 7 章曾讨论了嵌入模板，此处将采用类似的技术实现 pinboard 程序。对此，首先创建一个名为 index.html.js 的模板文件，并添加下列 HTML 元素。

```
//index.html.js
module.exports = `
<!DOCTYPE html>
<html>
  <head>
  <title>Pin board</title>
  </head>
  <style>
    body{
      background-color: #CCB;
    }
    .card {
      /* Add shadows to create the "card" effect */
      box-shadow: 0 4px 8px 0 rgba(0,0,0,0.2);
      transition: 0.3s;
      width: 30%;
      background-color: white;
      float: left;
      margin: 5px;
    }
    .card: hover {
      box-shadow: 0 8px 16px 0 rgba(0,0,0,0.2);
    }
    .container {
      padding: 2px 16px;
    }
    h4{
      text-align: right;
    }
    textarea{
      border: 0; padding: 10px; width: 90%; margin right: 5%;
      float: left;
      width: 60%;
    }
    input{
      float: left;
      width: 15%;
      padding: 15px;
    }
    button{
```

```
    padding: 15px;
    float: left;
    width: 20%;
  }
</style>
<body>
<textarea autocomplete="off" id="textData"></textarea><button
  id="postButton">Post</button>
  <div class="collection">
    <div class="card">
      <div class="container">
        <p>LoremIpsum is simply dummy text of the printing and
         typesetting industry. Lorem Ipsum has been the industry's standard
         dummy text ever since the 1500s, when an unknown printer took a
         galley of type and scrambled it to make a type specimen book. It
         has survived not only five centuries, but also the leap into
         electronic typesetting, remaining essentially unchanged. It was
         popularised in the 1960s with the release of Letraset sheets
         containing Lorem Ipsum passages, and more recently with desktop
         publishing software like Aldus PageMaker including versions
         ofLorem Ipsum.
        </p>
      <h4><b>John Doe</b></h4>
      </div>
    </div>
  </div>
</body>
</html>
`;
```

上述模板涵盖了 pinboard 应用程序所需的全部 UI 元素。其中，我们导出了一个包含所有 HTML 的简单字符串，例如，作为输入元素的 textarea；包含文本 Post（作为操作元素）的按钮，以及在插板上添加接口和卡片的 collection 类。接下来开始配置服务器，进而在浏览器请求处显示模板。

## 11.1.2　配置 Socket.IO 服务器

WebSocket 实现通过 Socket.IO 库进行封装和处理。Socket.IO 不仅是一个库，同时也是一个框架，从而提供了超出库范围的更多特性。下列命令可用于安装 Socket.IO。

```
npm install socket.io --save
```

当前，我们需要根据节点服务器的请求在浏览器中呈现模板。针对于此，可向 app.js 中添加下列代码片段。

```
const templateData = require('./index.html');
const server = require('http').createServer((req, res) => {
  res.setHeader('content-type', 'text/html');
  res.end(templateData);
});
const io = require('socket.io')(server);
io.on('connection', function(client) {
  client.on('disconnect', function() {
    console.log('user disconnected');
  });
  console.log("connected to realtime data server");
});
server.listen(3400);
```

当在浏览器中访问 http://localhost:3400 时，对应结果如图 11.1 所示。




图　11.1

下面将对上述代码进行逐一解释。此处导入了 index.html.js 模板，这是一个简单的节点模块，并可导出基于字符串的 HTML 数据。随后，我们创建了一个 HTTP 服务器，并

通过 Web 浏览器或其他显示 HTML 视图的客户端处理 HTTP 请求和响应。

在将服务器实例传递至所需的 Socket.IO 库后，此处创建了一个实时服务器。Socket.IO 是当前实时应用程序的原始库，它为 WebSocket 提供了一个包装器；而 WebSocket 是一种通过传输控制协议（Transmission Control Protocol，TCP）执行连续通信握手的协议。

简单地讲，一旦客户端构建了与 WebSocket 服务器间的连接，客户端每次不必生成特定的请求以获取特定的数据。一旦建立了对连接的第一个请求，客户端就可以接收与时间（循环和发送）和任何其他事件触发器相关的连续数据。客户端或服务器只需要触发一个事件来传递数据，并通过回调监听以接收数据。该事件模型通过发布-订阅模式加以使用，或者更具体地说，使用监听器。上述代码中设置了两个监听器，即 connection 和 disconnect。

通过下面的代码片段，我们将更准确地理解这些内容。在 app.js 中，将添加名为 connection 的新监听器，它将监听断开连接事件，如下所示。

```
io.on('connection', (client) => {
  console.log("connected to realtime data server");
  client.on('disconnect', () => {
    console.log("A user is disconnected!");
  })
});
```

### 11.1.3　配置 Socket.IO 客户端

通过添加一个客户端脚本，我们将被客户端 index.html.js 连接至 Socket.IO 服务器上，如下所示。

```
<script src="/socket.io/socket.io.js"></script>
<script>
const socket = io();
</script>
```

此处在<body>标签结尾后添加了<script>标签。socket.io.js 在调用 io 原型方法时处理与服务器的连接。默认状态下，它利用与提供给浏览器的套接字文件相同的 URL 建立连接；除此之外，它还提供了一个选项并可采用不同的方式设置连接 URL，即将其作为一个 URL 字符串参数进行传递。

在实现了 connection 和 disconnect 监听器之后，客户端也将被连接。当刷新浏览器时，将得到以下输出结果。

```
connected to realtime data server
A user is disconnected!
```

接下来将添加相关功能项，以便可捕捉用户输入内容，并将其置于 pinboard 上。这里将使用 jQuery 库，以简化与 UI 相关的代码。简单地讲，jQuery 是一个客户端-脚本 JavaScript 库，并可方便地操控 DOM 元素。对此，可通过下列代码包含 jQuery 库。

```
<script src="http://code.jquery.com/jquery-3.3.1.slim.min.js"
integrity="sha256-3edrmyuQ0w65f8gfBsqowzjJe2iM6n0nKciPUp8y+7E="
crossorigin="anonymous" />
```

注意，上述代码须置于自定义脚本标签之上，进而可访问 jQuery 的全局变量$，以供全部方法使用。

至此，我们对事件监听器的工作原理有了一定的了解。接下来将确定监听器的数量，以及根据何种条件满足下列决策。

（1）明确数据流的方向。对于单路或双路通信，这可以通过从浏览器文本区域捕获输入数据并将其发送到服务器来发现这一点。服务器可对数据执行持久化操作（当前仍为临时存储），并将其发送至所有已订阅的客户端（浏览器），这可通过下列代码加以描述。

```
//index.html.js
$(function() {
  const socket = io();

  $("#postButton").on('click', function(e) {
    let textData = $("#textData").val();
    let writtenBy = $("#writtenBy").val();
    socket.emit('new-pin', { story: textData, writtenBy:
    writtenBy });
    $("#textData").val('');
    $("#writtenBy").val('');
    return false;
  })
  socket.on('append-to-list', function(data) {
    console.log("data", data);
    $('.collection').append('<div class="card"><div
    class="container"><p>' +
    data.story + '</p><h4><b>' + data.writtenBy + '</b></h4>
    </div></div>')
  })
})
```

上述代码利用 postButton 的 id 在元素上绑定了一个单击事件，即一个 HTML 按钮。随后，我们使用 new-pin 作为通道，并从输入内容中提取数据，例如名称以及通过 WebSocket（ws://）协议发布和发出的文本。其间，预计将创建一个订阅器，并监听服务

器端的 new-pin 通道。

（2）下面实现 app.js 中的订阅器，如下所示。

```
let pinBoard = [];

io.on('connection', (client) => {
  console.log("connected to realtime data server");
  client.on('disconnect', () => {
    console.log("A user is disconnected!");
  })

  client.on('new-pin', (pinData) => {
    pinBoard.push(pinData);
  })
});
```

在上述代码中，可以看到较为清晰的变化内容。其中创建了一个临时存储数组 pinBoard，以存储供引用的全部数据。一旦接收了 new-pin 数据，即可将其推送至 pinBoard 中。需要注意的是，当前尚无法更新 UI。

（3）为了“记住”每一个订阅器和监听器，我们需要设置一个发射器（emitter）。当某个功能块丢失或无法正常工作时，将无法完成通信。对此，我们需要通过依次发送 pin 数据来更新 UI pin 列表，以便所有的订阅客户端均能够接收更新内容，如下所示。

```
client.on('new-pin', (pinData) => {
  pinBoard.push(pinData);
  io.emit('append-to-list', pinData)
})
```

这里创建了一个名为 append-to-list 新通道或命名空间。当在 append-to-list 命名空间中调用 client.emit 方法时，该方法将把数据发送至订阅者的监听器回调中。接下来，让我们在 index.html.js 的客户端代码上创建一个监听器，如下所示。

```
socket.on('append-to-list', function(data) {
  $('.collection').append('<div class="card">
  <div class="container"><p>' +
  data.story + '</p><h4><b>' + data.writtenBy +
  '</b></h4></div></div>')
})
```

如果向 pinBoard 中添加了新消息，下面将检测 pinBoard 是否在两个不同的浏览器选项卡中被更新。运行 node app.js 命令，并在浏览器中访问 http://localhost:3400，对应结果如图 11.2 所示。

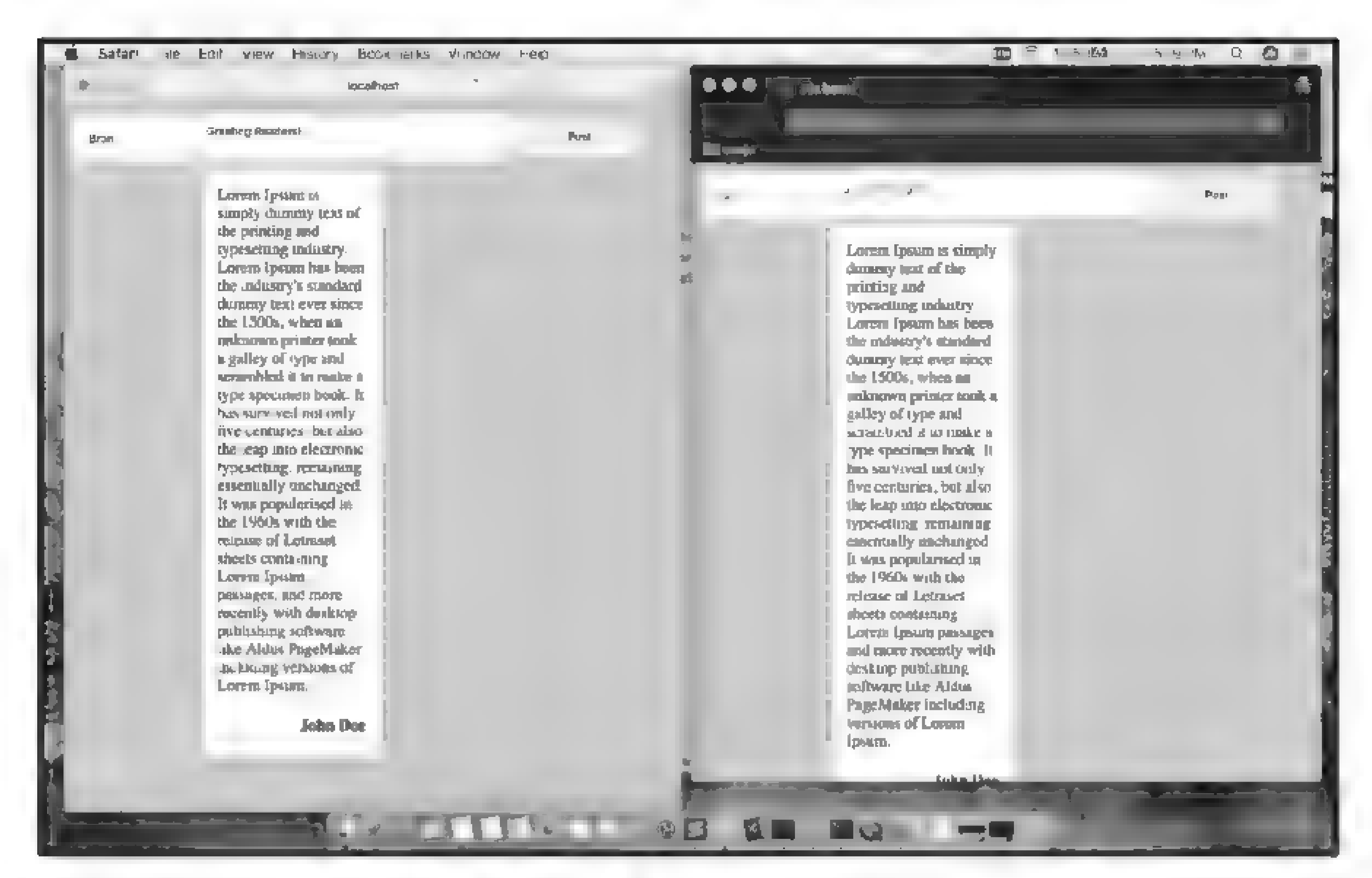



图　11.2

图 11.2 中并列显示了两个浏览器。此时，用户正在输入一个新帖子。这两个浏览器都允许客户端连接到 pinBoard 服务器应用程序。让我们在用户单击了 Post 按钮后检查客户端操作的结果。图 11.3 显示了单击后的状态。

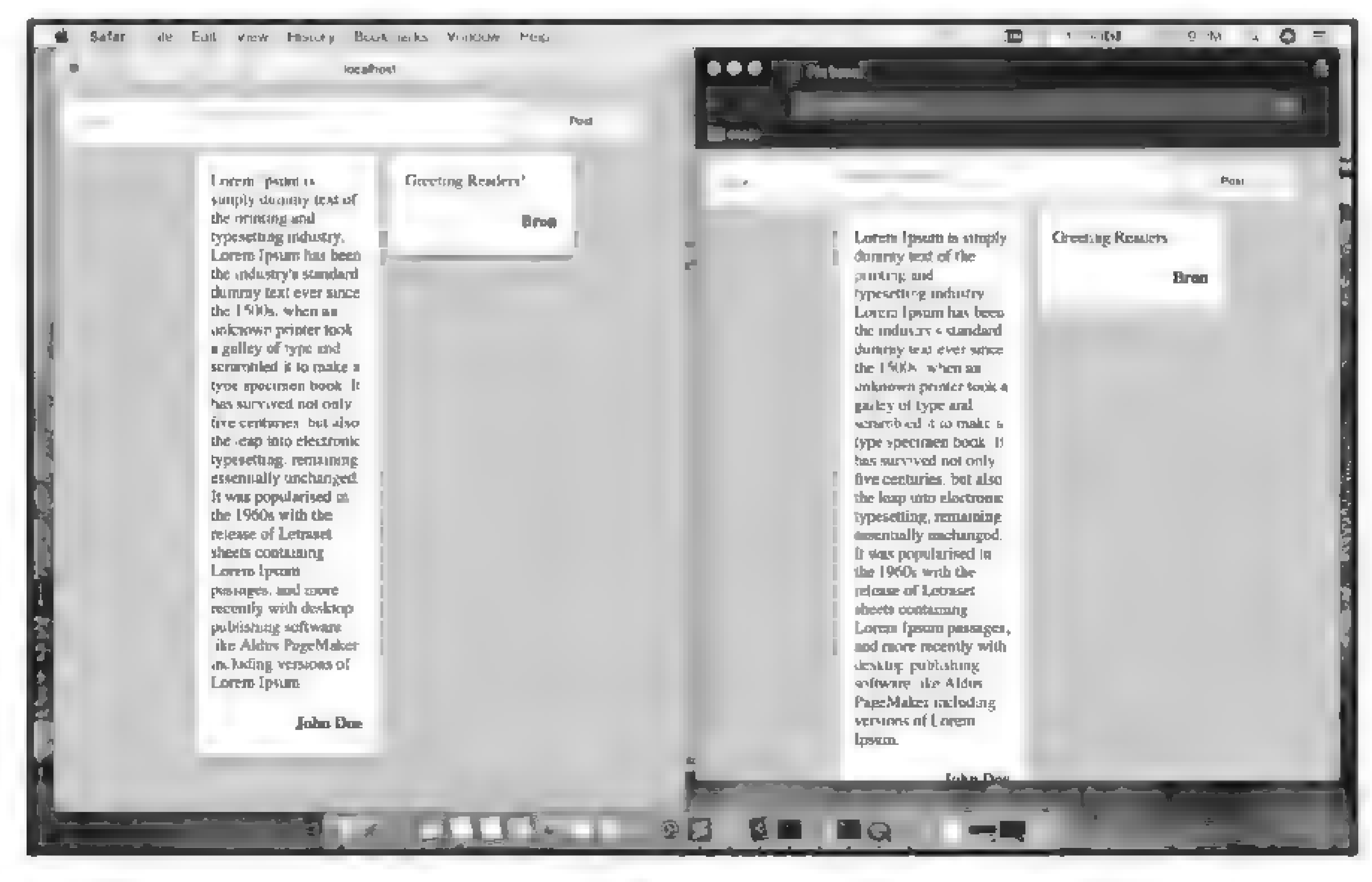



图　11.3

可以看到，两个浏览器客户端均接收到相同的帖子，这一结果十分有用。

（4）处理客户端默认的虚拟 pinBoard 数据，以及包含相关逻辑的服务器代码，如下所示。

```
io.on('connection', (client) => {
  console.log("connected to realtime data server");
  io.emit('pin-list', pinBoard)
  client.on('disconnect', () => {
    console.log("A user is disconnected!");
  })

  client.on('new-pin', (pinData) => {
    pinBoard.push(pinData);
    console.log("pinData", pinData);
    io.emit('append-to-list', pinData)
  })
});
```

在上述代码中，默认情况下，我们通过 pin-list 通道发送 pinBoard 数据。这样做是为了能够动态处理 pin-list 数据。

（5）在 index.html.js 中，移除将 card 作为类名的 div 元素及其所有内容，并将其移至 jQuery 代码中。这样，当在 pin-list 通道中接收时，仅当默认状态下 pinBoard 数据不存在时，方可将其追加至 collection 类元素中。下列代码显示了 index.html.js 中的修改内容。

```
//index.html.js
module.exports = `
<!DOCTYPE html>
<html>
<head>
  <title>Pin board</title>
</head>
<style>
body{
  background-color: #CCB;
}
.card {
  /* Add shadows to create the "card" effect */
  box-shadow: 0 4px 8px 0 rgba(0,0,0,0.2);
  transition: 0.3s;
  width: 30%;
  background-color: white;
  float: left;
```

```
  margin: 5px;
}
.card:hover {
  box shadow: 0 8px 16px 0 rgba(0,0,0,0.2);
}

.container {
  padding: 2px 16px;
}
h4{
  text-align: right;
}
textarea{
  border: 0; padding: 10px; width: 90%; margin-right: 5%;
  float: left;
  width: 50%;
}
input{
  float: left;
  width: 15%;
  padding: 15px;
}
button{
  padding: 15px;
  float: left;
  width: 20%;
}
</style>
<body>
  <input type="text" id="writtenBy" placeholder="name"/><textarea
  id="textData"
  autocomplete="off" placeholder="put your thoughts!"></textarea>
  <button
  id="postButton">Post</button>
  <div class="collection">
  </div>
</body>
<script
  src "http://code.jquery.com/jquery 3.3.1.slim.min.js"
  integrity="sha256 3edrmyuQ0w65f8gfBsqowzjJe2iM6n0nKciPUp8y+7E"
  crossorigin="anonymous"></script>
  <script src "/socket.io/socket.io.js"></script>
<script>
```

```
$(function () {

  const socket = io();
  $("#postButton").on('click', function(e){
    let textData = $("#textData").val();
    let writtenBy = $("#writtenBy").val();
    socket.emit('new-pin', {story: textData,
    writtenBy: writtenBy});
    $("#textData").val('');
    $("#writtenBy").val('');
    return false;
  })
  socket.on('append-to-list', function(data){
    $('.collection').append('<div class="card"><div
    class="container"><p>'+data.story+'</p><h4>
    <b>'+data.writtenBy+'</b></h4></div></div>')
  })
  /**
   * Pin-list get all the pins on load
   */
  socket.on('pin-list', function(list){
    console.log("list", list);
    if(list.length){
      list.forEach(function(data){
      $('.collection').append('<div class="card"><div
      class="container">
      <p>'+data.story+'</p><h4><b>'+data.writtenBy+'</b></h4>
      </div></div>')
      })
    }else{
      $('.collection').append('<div class="card"><div
      class="container"><p>Lorem Ipsum is simply dummy
      text of the printing and typesetting industry. Lorem Ipsum has been the
      standard dummy text ever since the 1500s, when an unknown printer took
      a galley of type and scrambled it to make a type specimen book. It has
      survived not only five centuries, but also the leap into electronic
      typesetting, remaining essentially unchanged. It was popularised in the
      1960s with the release of Letraset sheets containing Lorem Ipsum
      passages, and more recently with desktop publishing software like Aldus
      PageMaker including versions of Lorem Ipsum.</p><h4><b>John Doe</b>
      </h4></div></div>')
    }
  })
```

```
})
</script>
</html>
```

（6）最后，重新启动节点 App，并再次访问 http://localhost:3400。这里，输出结果保持不变，但 pin 列表已被正确地加以处理，如图 11.4 所示。

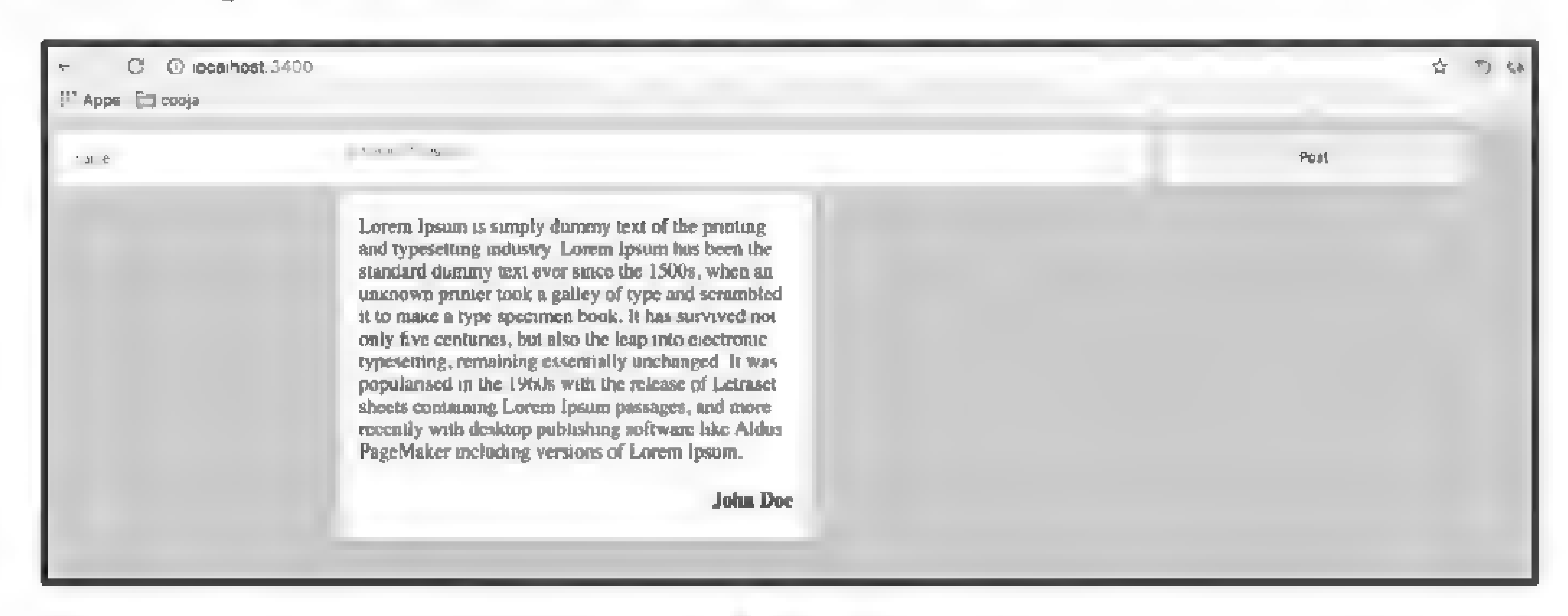



图　11.4

当前，基于 JSON 数据的实时应用程序已处于就绪状态。如果 WebSockets 未得到良好的支持，Socket.IO 可针对长轮询生成顺畅的回调，同时还可简化 WebSockets 的整体实现，并最大限度地利用服务器端和客户端。

至此，实时应用程序中的 JSON 实现暂告一段落。接下来将讨论分布式系统中的 JSON 实现。

## 11.2　在 Apache Kafka 中使用 JSON

分布式系统是在网络上隔离的逻辑系统。若应用程序希望在网络上实现横向伸缩，或者数据流随时间的变化而增长，则可考虑采用分布式系统。在后续内容中，数据流将简称为流。

Kafka 是一个分布式流处理平台，充当流的生产者和使用者代理。在 Kafka 中，生产者可视为提供数据的任何实体；而使用者则表示为接收数据的任何实体。

此类平台在股票市场和地理空间应用程序等领域十分有用，其间，数据是不断被产生和使用的。

本节将通过之前简单的实时应用程序来实现 JSON 数据的通信，进而研究 Apache Kafka 的各项功能。对此，图 11.5 展示了其中的一些关键字。

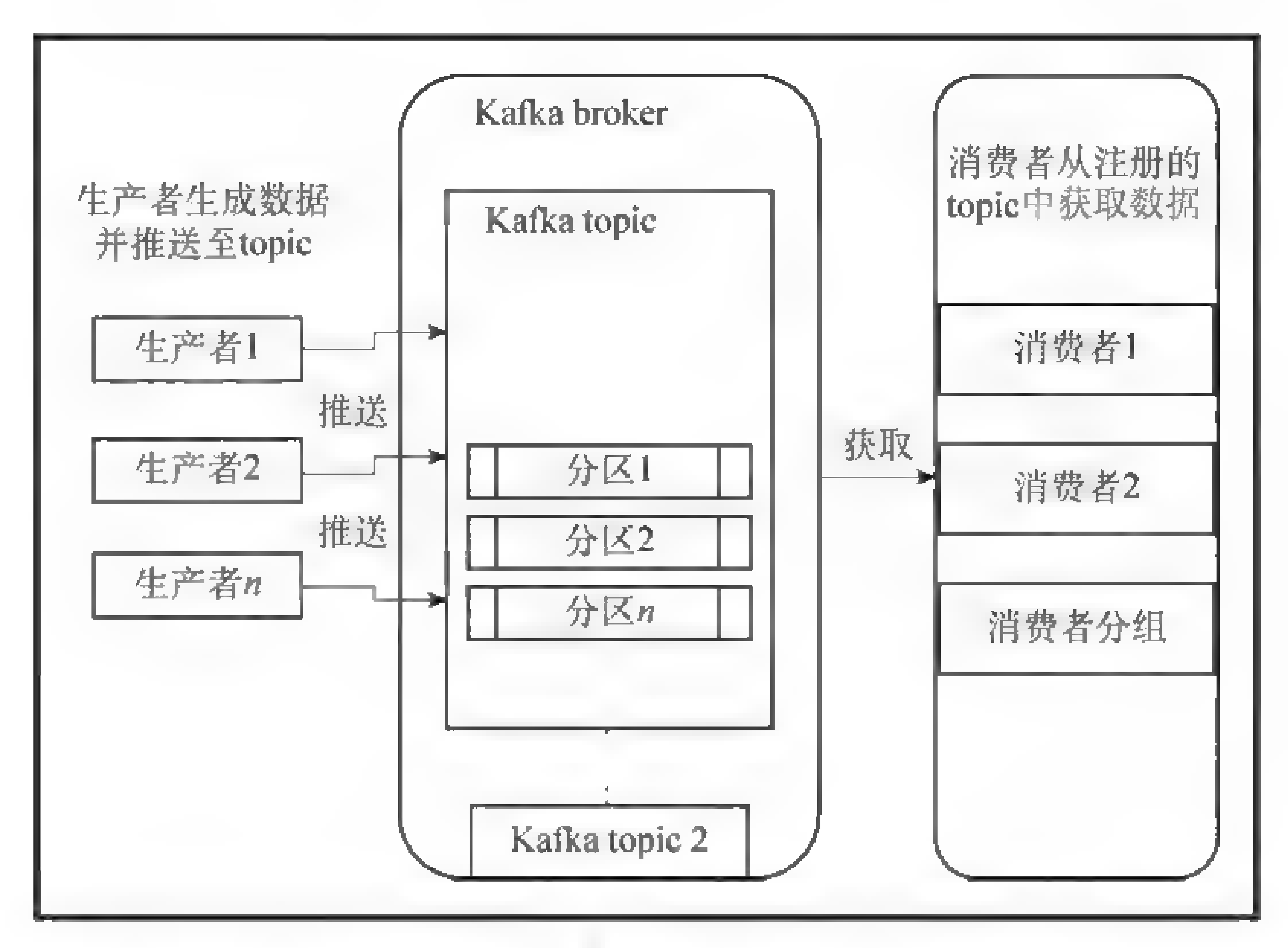



图　11.5

图 11.5 中显示了 Kafka 中一些基本的流。另外，Kafka 代理与 Zookeeper 实现了紧密的耦合，稍后将对 Zookeeper 予以介绍。

## 11.2.1　配置 Apache Kafka

读者可访问 http://kafka.apache.org/downloads 下载 Apache Kafka。需要注意的是，Unix 和 Windows 系统包含了不同的安装过程。

对于Unix 系统，需要解压 kafka_2.11-1.0.0.tgz 压缩包。当前下载的文件位于Downloads 目录中，我们可于其中使用下列命令完成处理过程。

```
$ cd ~/Downloads/
$ tar -xvf kafka_2.11-1.0.0.tgz
```

待全部文件解压完毕后，即可开始使用 Kafka。启动 Kafka 服务器的相关步骤如下。

（1）访问当前目录并执行下列命令。

```
$ cd kafka_2.11-1.0.0
```

（2）通过下列命令启用 zookeeper 实例。

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

确保 zookeeper 服务处于运行状态，且不会在各项步骤之间退出。图 11.6 显示了相应

的运行状态。

```
Last login: Fri Feb 23 11:43:05 on ttys007
brunos-MacBook-Pro:kafka_2.11-1.0.0 bruno$ bin/zookeeper-server-start.sh config/zookeeper.properties
[2018-02-24 07:26:40,552] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2018-02-24 07:26:40,554] INFO autopurge.snapRetainCount set to 3 (org.apache.zookeeper.server.DatadirCleanupManager)
[2018-02-24 07:26:40,554] INFO autopurge.purgeInterval set to 0 (org.apache.zookeeper.server.DatadirCleanupManager)
[2018-02-24 07:26:40,554] INFO Purge task is not scheduled. (org.apache.zookeeper.server.DatadirCleanupManager)
[2018-02-24 07:26:40,554] WARN Either no config or no quorum defined in config, running  in standalone mode (org.apache.zookeeper.server.quorum.QuorumPeerMain
)
[2018-02-24 07:26:40,576] INFO Reading configuration from: config/zookeeper.properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2018-02-24 07:26:40,577] INFO Starting server (org.apache.zookeeper.server.ZooKeeperServerMain)
```

图　11.6

（3）最后，通过下列命令在 Terminal 或命令行提示符中启用 Kafka 服务。

```
$ bin/kafka-server-start.sh config/server.properties
```

上述命令的输出结果显示了 config 目录中提供的配置日志，以及与 Zookeeper 间的连接状态。

**Zookeeper 是什么？为什么要使用 Zookeeper？它与纯 Kafka 有什么不同？**

相信读者也怀有此类疑问 为了实现应用程序的某些功能，在不同机器上运行的程序之间进行协调是一项困难的工作 Zookeeper 是一个开源服务供应商，它提供了分布式系统之间的简单协调机制 考虑以下场景：假设我们有两台服务器，一台 Zookeeper 的实例名为 broker 1，另一台 Zookeeper 实例名为 broker 2 如果 broker 1 在开始阶段即配置为主服务器，并且 broker1 出现故障，那么，Zookeeper 中央服务负责管理新代理的选取操作。

Kafka 自身是一个简单的消息传递系统，并提供了发布-订阅模型 此外，Kafka 也是一个基于临时数据库的队列系统 考查以下示例：假设需要开发一个系统并处理较为重要的非实时数据，例如电子邮件的发送 对此，可通过 Kafka 将作业推送至 Kafka 代理中；而 worker 程序则在需要时获取作业并发送电子邮件。这可视为一种基本思路。通过管理消息系统、实时流处理以及分布式系统协作，Kafka 与 Zookeeper 间的集成使其优势倍增因此，当讨论 Kafka 时，不可避免地会涉及 Zookeeper 集成，二者紧密耦合。另外，关于持久化存储，Kafka 则依赖于 Zookeeper 当利用相关命令启动 Kafka 时，同时也建立了与 Zookeeper 和所选主服务器间的连接。

相应地，Zookeeper 将管理 Kafka 集群，并跟踪主题、消息等内容 关于 Zookeeper 和 Kafka 的更多信息，读者可分别访问 http://zookeeper.apache.org/和 https://kafka.apache.org/。

## 11.2.2　利用 Socket.IO 应用程序实现 Kafka

本节将通过 Kafka 在实时应用程序中实现一个简单的 JSON 消息传递功能，并计划在某个特定的时间段内将随机消息发送至 Kafka 中。相应地，注册于 Kafka 中的所有使用者

均会接收到相关消息。作为一个接收者，我们还将使用实时 pinBoard 应用程序。

应用程序流和 Kafka 的一些细节内容如下。

（1）创建一个名为 kafka-app.js 的新文件，并隔离 Kafka 的最终实现版本。

（2）安装 kafka-node 作为 Kafka 的节点客户端。另外，这还将提供与 Zookeeper 之间的集成。下列命令用于安装 kafka-node。

```
npm install kafka-node --save
```

（3）包含 kafka-app.js，如下所示。

```
const kafka = require('kafka-node');
```

（4）作为依赖项启用 Kafka 代理进行消息传递，目前尚缺少一个步骤，即创建主题（topic）。因此，当连接 Kafka 实例时，还需要检查是否创建了一个名为 pinBoard 的主题。若不存在，下列代码将对此加以创建。

```
const kafkaClient = new kafka.Client();

kafkaClient.once('connect', function() {
  kafkaClient.loadMetadataForTopics([], function(error, results) {
    if (error) {
      return console.error(error);
    }
    let listofTopics = Object.keys(results[1]['metadata']);
    if (listofTopics.indexOf('pinBoard') == -1) {
      producer.createTopics(['pinBoard'], (err, data) => {
      console.log("New 'pinBoard' Topic created", err, data);
        //sendMessage();
      });
    } else {
      //sendMessage()
    }
  });
});
```

上述代码由以下活动构成。

（1）建立与 Kafka 间的连接。

（2）利用 kafkaClient.loadMetadataForTopics 获取与现有元数据相关的全部信息。

（3）如果所接收的响应中不存在 pinBoard 主题，则对其加以创建。

（4）创建一个生产者，并生成输入内容。我们有一个 JSON 文件，其中大约包含了 10 个输入，且每分钟被发送到 Kafka 中。下列代码用于包含该 JSON 文件。

```
const storyJSON = require('./story.json');
```

生产者函数如下所示。

```
function sendMessage() {
  let count = -1;
  setInterval(() => {
    count = count == 9 ? count = 0: ++count;
    /**
     * [messages multi messages can be an array,
     * single message can be a string or
     * a JSON]
     */
    producer.send([{
      topic: 'pinBoard',
      messages: JSON.stringify(storyJSON[count]),
    }], (err, data) => {
      console.log("Message send by producer", err, data);
    })
  }, 60000)
}
```

（5）当使用上述代码中的 producer 实例时，须在起始阶段的初始化过程中通过下列代码对其加以创建。

```
const producer = new kafka.Producer(kafkaClient);
```

（6）当 producer 实例可供使用时，可通过 producer.send()方法将消息数据发送至 Kafka 代理中。需要注意的是，这是一个异步方法，因而需要作为回调传递第二个参数。在 JSON 中的第 10 个元素之后，上述代码片段还设置了一条计数器重置逻辑。

另外，我们还需要对出现的错误进行处理。对此，Kafka 客户端模块提供了一个事件监听器，进而处理生产者-代理-使用者状态间出现的相关错误，如下所示。

```
producer.on('error', function(err) {
  console.log('Producer is in error state');
  console.log(err);
})
```

（7）取消对 sendMessage()调用的注释。除此之外，还应确保两个 Kafka 服务均处于运行状态；否则，可启动 Zookeeper 和 Kafka，并于随后通过下列命令运行 kafka-app.js 文件。

```
node kafka-app.js
```

对应输出结果如图 11.7 所示。

```
brunos-MacBook-Pro:chapter 11 bruno$ node kafka-app.js
Producer will send message at every interval of 1 min
 Waiting for 1 min...
Message send by producer null { pinBoard: { '0': 23 } }
```

图　11.7

通过上述各项步骤，我们已经成功地创建了一个 Kafka 生产者，并将数据保存至 Kafka 中。当检测实时更新以及消费者是否接收数据时，可利用下列 shell 命令连接 Kafka 消费者。

```
$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092
--topic pinBoard --from-beginning
```

如果 Kafka 中包含了 Zookeeper 的早期版本，应确保在上述消费者配置命令的执行过程中传递强制参数--zookeeper <urls>。其中，urls 由主机和端口组成，进而提供了 Zookeeper 连接。相应地，还可包含多个 URL 以处理代理故障

上述命令的输出结果如图 11.8 所示。

```
brunos-MacBook-Pro:kafka_2.11-1.0.0 bruno$ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic pinBoard --from-beginning
{"story":"You cannot shake hands with a clenched fist","writtenBy":"Indira Gandhi"}
{"story":"When you reach the end of your rope","writtenBy":"tie a knot in it and hang on"}
{"story":"Learning never exhausts the mind","writtenBy":"Leonardo da Vinci"}
{"story":"You cannot shake hands with a clenched fist","writtenBy":"Indira Gandhi"}
{"story":"When you reach the end of your rope","writtenBy":"tie a knot in it and hang on"}
{"story":"No act of kindness, no matter how small","writtenBy":"Aesop"}
{"story":"When you reach the end of your rope.. tie a knot in it and hang on","writtenBy":"Franklin D. Roosevelt"}
{"story":"But man is not made for defeat. A man can be destroyed but not defeated","writtenBy":"Ernest Hemingway"}
{"story":"When you reach the end of your rope.. tie a knot in it and hang on","writtenBy":"Franklin D. Roosevelt"}
{"story":"There is nothing permanent except change","writtenBy":"Heraclitus"}
{"story":"You cannot shake hands with a clenched fist","writtenBy":"Indira Gandhi"}
{"story":"Learning never exhausts the mind","writtenBy":"Leonardo da Vinci"}
```

图　11.8

接下来实现实时应用程序中“消费者”这一概念，这也将利用 Kafka 完成我们的 JSON 消息传递系统。这里，app.js 视为当前的使用者，其中包括：

（1）首先需要创建一个消费者，并在端口 9092（这也是默认端口）上连接至 Kafka 客户端，如下所示。

```
const kafka = require('kafka-node'),
  client = new kafka.Client(),
  consumer = new kafka.Consumer(client,
    [{ topic: 'pinBoard', offset: 0 }],
    {
      autoCommit: false
    }
  );
```

这里创建了一个消费者实例，它采用 topic 连接至 Kafka。此类主题作为数组参数传递至 kafka.Consumer()方法中。

（2）创建一个错误处理程序，并管理消费者一端的全部错误，其实现过程如下所示。

```
consumer.on('error', function (err) {
  console.log('Error:',err);
})
```

（3）一旦 consumer 准备就绪，即可监听消费者获取的全部消息。注意，在 Kafka 中，消费者获取消息意味着 Kafka 服务并不像 ping-pong 服务器那样显式地发送消息。这也是实时实现中的不同之处，像是在一个 Socket.IO 应用程序中。

对应代码如下所示。

```
consumer.on('message', function(message) {
  console.log("consumer message-->", (message));
  if (typeof message.value == 'string') {
    const pinData = JSON.parse(message.value);
    pinBoard.push(pinData);
    io.emit('append-to-list', pinData)
  } else
    throw message.value;
});
```

上述代码用 consumer.on('message',function(message) {})事件监听器方法获取消息，当所接收的数据是字符串时将采用 JSON.parse 解析消息。鉴于 JSON 的结构等同于 pinBoard 应用程序中 pin 的 JSON，因而可向 append-to-list Socket.IO 通道发送 pinData，以使每分钟后可在浏览器中查看到新的 pin。

（4）通过下列命令启动使用者节点服务。

```
node app.js
```

（5）连接至 http://localhost:3400/，对应结果如图 11.9 所示。

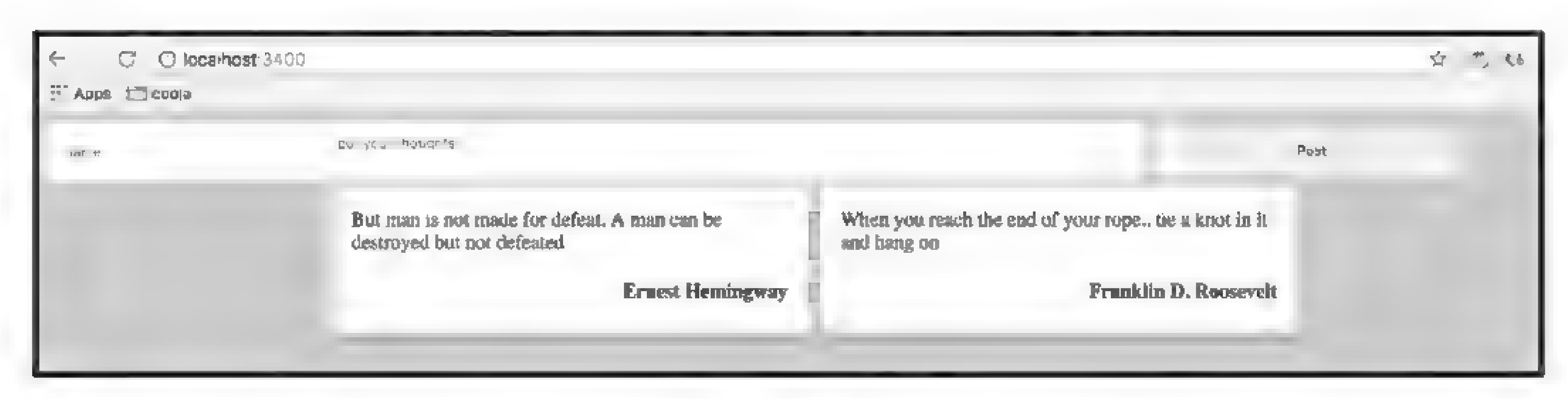



图　11.9

在图 11.9 中可以看到，每分钟将有新内容添加至列表中。

需要注意的是，我们可将多个消费者应用程序连接至运行于不同端口上的 Kafka 客户端，进而接收来自生产者的 Kafka 消息。另外，根据 Kafka 集群，生产者还可以作为消费者进行扩展。唯一需要注意的是保持维护的成本。

## 11.3　本 章 小 结

本章针对实时系统和分布式系统实现了 JSON 数据。其中，关于如何将数据作为集合传递给 WebSocket 通道，实时实现给出了相应的整体思路。WebSocket 实现于 Socket.IO 中，其速度非常快且工作可靠，它目前在产品级别上服务于大量的应用程序，例如 Trello、Blog Talk Radio 和 Zendesk。

另外，本章还介绍了分布式系统中的一些概念，并实现了 Kafka 中的简单功能。基于实时应用程序的 Kafka 实现提出了这样一种理念，即扩展像消费者一样工作的实时应用程序。

了解实时和分布式系统的概念将有助于读者为数据可用性和系统可伸缩性提供解决方案。第 12 章将讨论不同领域中实现 JSON 数据时的高级 JSON 格式，例如地理空间领域、SEO 和数据存储。

# 第 12 章　JSON 中的用例

JavaScript 简单对象表示法是改变数据交换技术领域的一颗种子。简单的大括号（{}）格式、键-值对结构、可读性和易操作性使它具有很大的吸引力。事实上，JSON 已经成为该领域的领先者，其他数据交换格式都以 JSON 为基础。开源生态系统中存在多种 JSON 格式，且均源自 JSON。

本章主要考查以下几种格式。

- GeoJSON。
- JSONLD。
- BSON。
- MessagePack。

下面将对此加以逐一讨论。

## 12.1　GeoJSON——地理空间 JSON 数据格式

GeoJSON 是基于 JSON 的一种实现，同时专门针对地理空间数据而设计。这里，地理空间数据是表示任何空间或其几何形状的一个区域的信息。

区域的形状可以可视化为正方形、矩形或具有特定测量值的任何其他多边形，并使用纬度和经度坐标进行定位。

GeoJSON 由互联网工程任务组（Internet Engineering Task Force，IETF）予以标准化。GeoJSON 针对各类地理空间数据提供了相关规范。下面考查基本的 geoJSON 结构，如下所示。

```
{
 "type": "Polygon",
 "coordinates": [
   [
     [100,0],
     [101,0],
     [101,1],
     [100,1],
     [100,0]
   ]
  ]
}
```

上述 JSON 数据设置了两个键，对应类型表示为几何类型，即引用了 7 个大小写敏感的字符串：Point、MultiPoint、LineString、MultiLineString、Polygon、MultiPolygon 和 GeometryCollection。

接下来是坐标键，它是 lat 和 long 值对构成的数组点列表。每组数值形成了几何类型的地理形状。

每个几何对象都是一个 Feature 对象的包装器。Feature 对象包含几何图形和属性，用户可以将这些几何图形和属性设置为 JSON、null 或其他可能需要的格式，如下所示。

```
{
 "type": "Feature",
 "geometry":
 {
   "type": "LineString",
   "coordinates": [
     [102, 0],
     [103, 1],
     [104, 0],
     [105, 1]
   ]
 },
 "properties":
 {
   "prop0": "value0",
   "prop1": 0
 }
}
```

当某个应用程序需要使用多个 Feature 列表时，geoJSON 还设置了一个名为 featureCollection 的数组结构。featureCollection 由多个 Feature 构成。

这种媒体类型可应用于多种领域，例如 Web 映射、地理空间数据库、地理数据处理 API、数据分析和存储服务以及数据传播。

目前，市场上已出现了基于 geoJSON 的多种工具，例如 leaflet.js（参见 http://leafletjs.com）、cartodb（参见 https://github.com/CartoDB/cartodb）以及 turf.js（参见 http://turfjs.org/）。

## 12.2　JSONLD——针对 SEO 的 JSON 格式

JSONLD 是指包含链接数据结构的 JSON。下面首先讨论链接数据结构的含义。假设我们在块链上编写一个博客，我们需要向 Web 爬虫程序提供一些元数据，假设这里采用

的是谷歌爬虫程序。爬虫程序可解析 HTML 并读取博客内容。当前，此类爬虫操作通常由机器完成，而非手工操作。这些输入元数据可能包含一些与下一篇或上一篇博客文章相链接的信息，也可能包含与发布该博客的用户相链接的信息。针对于此，JSONLD 提供了一种方法可实现这两种操作。也就是说，它既负责处理爬虫程序的元数据，也提供了其他数据集的链接。

例如，当搜索区块链时，谷歌搜索引擎使用抓取的数据提供最佳的搜索结果。

JSONLD 可用于连接 Web 上的数据。接下来将进一步理解 JSONLD，并与链接数据协同工作。

在 Web 开发过程中，JSONLD 在 HTML<script>标签中一般定义为 type="application/ld+json"，如下所示。

```
<script type="application/ld+json">
  {JSONLD data}
</script>
```

除非在<script>标签中传递类型为 application/ld+json 的 JSONLD，否则搜索引擎不会识别它。

一旦<script>标签就绪，即可将链接的 JSON 数据分配至当前上下文中。对此，需要定义 JSON 数据的上下文环境。该上下文可以是任何事物，并向实体提供背景词汇表。如果实体是一名作者，那么背景词汇表可包含名称、地址、书籍和发布日期等。当前上下文的定义方式如下所示。

```
<script type="application/ld+json">
{
  "@context": "http://schema.org/Person"
}
</script>
```

在上述 script 标签中，上下文将链接至 schema.org，特别是将与某个 person 实体链接。这里，schema.org 具体是指：

“Schema.org 是一个具有协作性质的社区活动，其任务是为互联网、Web 页面、电子邮件消息上的结构化数据创建、维护和推广相关模式”。

——schema.org

如果任何站点提供了与上下文相关的结构化数据，我们可对此加以使用。对此，读者可访问 https://json-ld.org/contexts/person.jsonld 以查看另一个示例。

下一个较为重要的键是@type。在当前示例中，可直接将其指定于@context 中（表示为一个 person）。相应地，我们可利用@type 关键字分别对其加以定义，如下所示。

```
<script type="application/ld+json">
{
  "@context": "http: //schema.org",
  "@type": "person",
  "name": "robin sharma",
  "work": {
    "@type": "CreativeWork",
    "About": "books,
    public speaking"
  }
}
</script>
```

这种不同的定义方式也使得我们可针对多种类型使用上下文。在当前示例中，当定义与 person 相关的数据时，我们采用了@type。对应类型提供了更为丰富的属性，例如名称和地址。所有属性（例如 name 和 about）都与@type 相关。

尽管 JSONLD 已经使用了一段时间，但许多网站并没有充分地利用它。在考虑如何获得最佳的 SEO 结果时，JSONLD 则变得非常重要。

## 12.3　BSON——快速遍历的 JSON 格式

前述章节曾讨论了 MongoDB 中的 BSON 实现。实际上，MongoDB 是第一个充分利用 BSON 的数据库。本书并不打算介绍其他实现方式，此类内容超出了本书的讨论范围。感兴趣的读者可访问 http://bsonspec.org/implementations.html 以了解更多内容。

由于 JSON 数据将转换为二进制数据，因而 BSON 数据的遍历速度将会有一定的提升。通过对 binData 和 Date 类型提供更为丰富的数据类型方面的支持，BSON 扩展了 JSON 数据类型集合。这一优点也使得 BSON 可用作任何非结构化数据库设计中的记录。

如果 BSON 格式用于网络上的数据传递，考虑到数据的二进制特征，不同网络设备间的编码和解码将更加简单，其处理速度也将有所提升。

BSON 具有存储效率——但是，与序列化的 JSON 相比，其尺寸可能稍微大一些，其原因在于，BSON 数据涵盖了更多属性，例如针对数据所呈现的数据类型和数据长度。

## 12.4　messagePack

需要说明的是，还有一种数据交换格式，它比我们目前介绍的所有数据交换格式都更节省空间。另外，它还提供了二进制序列化数据，旨在实现内存中的快速操作。许多著名

的站点，例如 Pinterest，都采用了 messagePack 来压缩数据。

考查以下场景：我们需要在主缓存（例如 Redis）中存储某些数据。在这种情况下，每次开发人员将数据存储至与某个键（在 Redis 中，数据存储表示为单层深度的键-值对）关联的单一字符串时——可能是 JSON、一个数组或者任何其他数据类型，随着数据的不断插入，内存利用率也会随之增加。对此，messagePack 将有助于减少内存的使用率。当采用 messagePack 编码数据时，将产生大约 40%的无损压缩。

messagePack 的作者所建议的另一个应用程序是跨服务 RPC 通信协议。在这种情况下，两个不同的应用程序进程希望进行通信，跨进程传递的数据可能需要一些额外的实现开销，而处理和存储则会面临 Endianness 问题。我们可以使用 messagePack 作为 RPC 的公共通信协议。

与运行在谷歌 v8 编译器上的普通 JSON 解析器相比，messagePack 的速度稍逊一筹。v8 利用字符串编排机制提供了高度优化的结果。有人曾用 v8 引擎上的 JSON 编码器对 messagePack 库进行了基准测试，对应结果显示于 https://github.com/mattheworiordan/nodejs-encoding-benchmarks 上。

## 12.5　本 章 小 结

在本章中，我们学习了一些 JSON 格式的案例研究。每个案例都具有自己的优点和具体含义，读者在为其实现作出合理决策时需对此有所了解。这里，也希望读者在学习 JSON 的过程中度过了愉快的时光。但这并不是终点——新的旅程现在已经启动！我们也希望读者能够从小处着手，不断地练习，因为长期的努力比短时的冲动更为重要。